UiO : Department of Informatics
University of Oslo

The CERES project

A Cloud Energy Reduction System

Thomas Hage

Master's Thesis Spring 2014

The CERES project

Thomas Hage

20th May 2014

Abstract

Cloud environments at data centers in general have well established architectures for uninterrupted power, security, expansion opportunities and availability. However, the architectures are generally consuming much more energy than required.

The problem statement is addressed by developing a prototype to demonstrate how energy efficiency can be optimized. This is done by using three multi dimensional bin packing algorithms. The CERES project addresses both the algorithms for scaling physical compute nodes in a cloud environment and the practical use of algorithms from a system administration perspective.

Our results and experiences show that each algorithm has different effect on the balance between energy efficiency and performance. The simplest bin packing algorithm provides useful mechanism to reduce power consumption, while more complex algorithms do not give the same power saving, but give better performance.

The project was able to optimize the energy efficiency by lowering the energy consumption with 40% without reducing the performance. An even higher reduction can be achieved by overbooking the hardware capacity given to the virtual machines. Some experiments did reduce the consumption with as much as 72%.

Contents

1 Introduction **1**

2 Background **5**

2.1 Virtualization and cloud computing 5
2.1.1 Types of virtualization 5
2.2 Hypervisors 6
2.3 Cloud computing 6
2.3.1 Data centers and Cloud providers 8
2.4 Cloud Platforms 8
2.5 OpenStack 10
2.5.1 Architecture of OpenStack 10
2.5.2 Compute(Nova) 10
2.5.3 Networking(Neutron) 11
2.5.4 Storage(Swift/Cinder) 11
2.5.5 OpenStack with KVM and libvirth 11
2.6 Manage Large Networks - MLN 11
2.7 Openstack Database 12
2.8 Energy efficiency in data centers 12
2.8.1 Live migration of virtual machines 13
2.9 Dynamic scaling in cloud computing 14
2.9.1 Relevant research in power-aware scaling 15
2.9.2 Service level scaling 15
2.9.3 Relevant research in performance application scaling . . . 16
2.9.4 Hardware scaling 16
2.10 Similarities in the approach 17
2.10.1 Energy consumption and CPU usage 19
2.11 Bin packing 20
2.12 Relevant work in bin packing projects 21
2.12.1 BtrPlace 21
2.12.2 Space Defragmentation Heuristic for 2D and 3D Bin Packing Problems 22
2.12.3 Dependable Virtual Machine Allocation 22
2.13 Tools for modeling and implementation 22
2.13.1 Dell Remote Access Controller - DRAC 22
2.13.2 Perl scripts 23
2.13.3 Storing the data - OpenTSDB 23
2.13.4 Sensu 23

2.13.5 Pseudocode . . . 23
2.13.6 Basic Representation of Interactive Components - BRIC . 23

3 Approach **25**
3.1 Objectives . . . 25
3.2 Understanding the complexity of a working framework . . . 27
3.3 Design phase . . . 27
3.3.1 Modeling . . . 28
3.3.2 Policies . . . 29
3.4 Implementation . . . 29
3.4.1 Environment . . . 30
3.4.2 Prototype . . . 30
3.4.3 Data collection . . . 30
3.5 Experiments . . . 31
3.5.1 Tasks for testing the environment . . . 31
3.5.2 Experiments for policies . . . 31
3.6 Comparison of the algorithms . . . 32
3.7 Long term testing . . . 32
3.8 Expected results . . . 33
3.9 Appraising properties . . . 33
3.9.1 Autonomic Computing . . . 34
3.10 Measurements and Analysis . . . 34

4 Result 1 - Design and models **35**
4.1 Model overview . . . 35
4.2 Model of the prototype . . . 35
4.2.1 The modules in the prototype . . . 36
4.3 Applying bin packing to virtual machines . . . 38
4.4 Algorithm I: First fit bin packing . . . 38
4.4.1 Formal notations . . . 39
4.4.2 The algorithm - policy . . . 41
4.4.3 Example . . . 43
4.5 Algorithm II: 2D best fit bin packing with CPU zones and minimal migrations . . . 45
4.5.1 Formal definitions . . . 46
4.5.2 The algorithm - policy . . . 46
4.5.3 Example . . . 49
4.6 Algorithm III: 3D bin packing with importance as constraint . . . 52
4.7 The environment . . . 54

5 Result 2 - Implementation for experiments **57**
5.1 Implementation - Building the system . . . 57
5.1.1 Monitoring power consumption . . . 57
5.1.2 Monitoring of OpenStack - VCPU and Memory usage . . . 60
5.1.3 Building the masterscript . . . 61
5.2 Policy 1 - Algorithm I: Simple first fit bin packing . . . 63
5.3 Policy 2 - Algorithm II: 2D best fit bin packing with CPU zones and minimal migrations . . . 66

5.4 Policy 3 - Algorithm III: 3D bin packing with importance as constraint . . . 69

6 Result 3: Measurements and analysis **73**
6.1 Testing the environment and prerequisites . . . 73
6.2 The experiments of policy 1 . . . 74
6.3 The experiments for policy2 . . . 77
6.4 The experiments for policy3 . . . 80
6.5 Comparing the three algorithms . . . 82
6.5.1 Policy 1 . . . 84
6.5.2 Policy 2 . . . 84
6.5.3 Policy 3 . . . 85
6.6 Power consumption of a fully packed compute node . . . 86
6.7 Temperature changes of a fully packed compute node . . . 87
6.8 Long term testing . . . 88
6.9 Burst of virtual machines . . . 92

7 Discussion: Managing a scaling cloud **95**
7.1 The model . . . 95
7.1.1 Implementation . . . 95
7.2 The difference between the policies and comparison with other research projects . . . 96
7.2.1 Long-term testing . . . 97
7.2.2 Avoiding live migrations . . . 98
7.2.3 Target nodes and power consumption . . . 99
7.2.4 Adding extra capacity . . . 99
7.2.5 Algorithm intervals . . . 99
7.2.6 Monitoring and other scheduled tasks . . . 100
7.2.7 Levels of importance . . . 100
7.2.8 Alternative approaches . . . 101
7.3 Future work . . . 102

8 Conclusion **105**

9 Appendix **111**
9.1 Policy 1 . . . 111
9.2 Policy 2 . . . 118
9.3 Policy 3 . . . 128
9.4 CPU affinity script . . . 136
9.5 Monitoring script for Count migrations . . . 144
9.6 Monitoring script power consumption . . . 147
9.7 Monitoring script for virtual machines . . . 149
9.8 Monitoring script for compute nodes . . . 152

List of Figures

2.1 Three types of virtualization 6
2.2 Cloud services(SaaS,PaaS,IaaS) 7
2.3 VMware logo 9
2.4 OpenStack logo 9
2.5 Amazon logo 9
2.6 Openstack architecture 10
2.7 A live migration 14
2.8 The process of binpacking 20
2.9 A completed bin packing 20
2.10 A BRIC model example 24

3.1 Graphical illustration of the approach 26
3.2 The design phase 28

4.1 Model of the prototype 36
4.2 BRIC model of the CERES prototype 37
4.3 Illustration of environment 39
4.4 The start of policy 1 41
4.5 BRIC model of policy 1 42
4.6 Illustration of environment 43
4.7 Policy 1 completed 44
4.8 Policy 2 dividing CPU affinity 45
4.9 Policy 2 migrating virtual machines 47
4.10 Policy 2 packing important and not important VM 47
4.11 BRIC model of policy 2 48
4.12 The start of Policy 2 49
4.13 Policy 2 sorted according to importance level 51
4.14 Policy 2 completed 52
4.15 Policy 3 bin packing 53
4.16 Policy 3 packing virtual machines 54
4.17 Hardware architecture 55
4.18 Hardware rack architecture 56

5.1 Compute nodes running and power consumption 59
5.2 BRIC model of policy 1 66

6.1 Policy 1 results with VCPU free per node 75
6.2 Policy 1 results running compute nodes 76

6.3 Policy 2 results number of running VM per node 78
6.4 Policy 2 result live migrations 79
6.5 Policy 2 resulting compute nodes 80
6.6 Policy 3 result of compute nodes running 81
6.7 Policy 3 resulted in this number of live migrations 82
6.8 Comparing the number of compute nodes and live migrations . . . 83
6.9 Comparing the power reduction 84
6.10 Power consumption while node fully packed 87
6.11 Temperature while node fully packed 88
6.12 The start of long term test 89
6.13 Power consumption during live migrations 90
6.14 Running compute nodes during long term test 90
6.15 Graph of running nodes and number of virtual during long term test 91
6.16 Live migrations during the long term test 91
6.17 The burst of live migrations 92
6.18 The VCPU free per node during burst 93
6.19 Free VCPU cores during burst and policy 2 94
6.20 Policy 2 during after burst of machines 94

List of Tables

6.1 Results of the simulations of policy 1 84
6.2 Results of the simulations of policy 2 85
6.3 Results of the simulations of policy 3 86
6.4 Power consumption of fully packed compute node 86
6.5 Results for the long term test of policy 2 92

Acknowledgments

First I would like to express my sincere gratitude and appreciation to my supervisor, Kyrre Begnum for his dedication and enthusiasm. His guidance and support throughout the master program has made this project possible, and made the work interesting, fun and educational. Secondly I would like to thank Anis Yazidi for his support during the time of research and contribution to this project.

Thanks to Oslo and Akershus University College and the University of Oslo for giving me the opportunity to take part in this master program and a special thanks to all the lectures, professors and fellow students for their guidance and inspiration to help me complete the master program.

I also wish to express my appreciation to my dearest family and friends for all the encouragement and mental support during the thesis. Finally genuine and heartfelt thanks to my girlfriend, Ane for all patience, love and support during the master program.

Sincerely,
Thomas

Chapter 1

Introduction

Internet-based computing is rapidly expanding and the electricity demand of cloud computing is now ranked as the 6th in the world compared to countries. Cloud computing consumes more energy every day than countries like Germany, Canada and Brazil [30]. The number of data centers has increased with 56 percent worldwide from 2005 to 2010 [18] and more hardware is installed to handle the increase of demand. The energy footprint of data centers is the fastest growing and will increase 81% by 2020, according to Green Peace [30].

In essence, a cloud is nothing more than hundreds of machines working together in one or more data centers, and these machines, called compute nodes are working together to make the cloud appear as one single system. By combining all these machines the cloud becomes highly complex, which makes it difficult for data centers to optimize the efficiency. On the other hand centralizing resources gives the possibility to measure and track the energy use and an average data center is wasting as much as 90 percent of the electricity powering idle servers [34].

The role of system administrators today is not only to provide data centers uninterrupted power, better security, expansion opportunities and availability, but to do this in a way that optimizes energy efficiently. Different approaches exist but they often involve a manual procedure done by the administrator. Most common is to either reduce power consumption by controlling the usage of electricity or to balance computing capacity after demand. Both solutions have their drawbacks in terms of performance and configurability. Most research in this field base their results on simulations or test environments. No research is found that attempts power saving for large cloud environments based on any of the major private cloud solutions, like OpenStack, Eucalyptus, OpenNebula or CloudStack. What is it like to run a scaling cloud in production? What practical details are overlooked by the theoretical models?

The aim of this paper is to help answer these question through the definition and implementation of three bin packing algorithms for scaling the physical compute nodes in a cloud environment. We are interested in not only the algorithm itself, but the practical implications from a system administration perspective of letting go of the control of the cloud and see it dynamically change the number of com-

pute nodes as it sees fit. Our assumption is that a cloud administrator needs the capability to shrink the cloud to a minimum based on defined constraints at certain points where energy conservation is the main goal. Our approach complements work on workload optimization and load balancing for active periods by providing an alternative for non-active periods.

More research on how resources can be managed efficiently is desirable and the aim is to build an advanced model combined with policy, to autonomically move virtual machines before dynamically scaling idle hardware according the user workload. The complexity of a cloud prevents data centers to scale the physical compute nodes in a cloud environment and there are several practical challenges associated with dynamic scaling of hardware customized for cloud computing.

The potential gain is research for a more energy efficient cloud, which can save both operation cost for the data center and for the customers. Idle servers in private data centers are wasting a substantial amount of the electricity and by knowing that the consummation of energy is higher than Brazil, there will be much to save both for the environment and for the owner and customers of data centers.

Problem statement *How can energy efficiency be optimized by dynamic scaling of the compute nodes in a cloud environment?*

Energy efficiency used in the problem statement refers to the amount of energy a data center is using, and how much of this energy that is for something productive. This can be divided by the measure "power usage effectiveness"(PUE) which is a calculation of how much of the total energy that goes to the computing equipment and not to other data center facilities like cooling and lighting. Data center infrastructure efficiency (DCIE) is a measure for how efficient the infrastructure is in comparison of totalt amount of energy consumed by the data center. In this case energy efficiency refers to making the most of the energy that is consumed and to lower the required amount of energy according to the performance demands in the data center. The purpose is to lower the amount of required compute nodes and therefore also decrease the power consumption.

The term *optimize* is a widely used concept in the field of computer science and is often used to describe the process of improving a program or infrastructure to make it run faster, perform better and to take full advantage of the resources that are available like time, cost and storage capacity. This is often a compromise among several of these conflicting requirements, and can result in an advanced engineering design.

One of the main challenges of a data center is idle hardware that consume more than necessary electricity. *Dynamic scaling* is a term indicating a specialized software that can take intelligent decisions based on a overview and status of all machines running in the environment to see if it is possible to adjust the number of physical machines according to user demand. Dynamic is an important term

because it can include so many features like automation and decision making processes which are key features in a complex environment that should not be manually done by system administrators manually.

Cloud computing is a collection of variations of computing concepts distributed over a network. Cloud computing provides the possibility to offer software, infrastructure, platforms as a service over network.

Chapter 2

Background

This chapter will introduces several technologies, concepts and applications which will be used in later chapters.

2.1 Virtualization and cloud computing

Virtualization was introduced in the 1960‘s by International Business Machine(IBM) and makes it possible to have multiple machines sharing the same hardware. Virtualization is the process where a machine called the "host" shares computing resources with a "guest" machine. The host controls all physical resources and allocates available resources to guest machines often called virtual machines. These virtual machines can have different operating systems from the "host" and are unaware of other machines located at the same "host" and the "host" it self. A hypervisor also called virtual machine manager is the software that controls all the physical hardware at the host [9] [44].

2.1.1 Types of virtualization

There are three main types of virtualization. These are partial virtualiazation, full virtualization and para virtualization.

Partial virtualization

Partial virtualization is often called operating system virtualization because it runs on top of the already installed host operating system. A virtualization software running on the "host" machine provides resources to "guest" operating systems and creates a illusion that the machine is running directly on the hardware. Each virtual machine is separated both from other virtual machines and from the "host". This type of virtualization gives advantages both in form of a simple installation and opportunity to keep systems or software separate on one single machine which can enhance security and flexible provisioning. However, the drawback is performance and is therefor mostly used on personal computers [43].

Full virtualization

Full virtualization is made possible through a "hypervisor" which replace the "host" operating system in partial virtualization. This improves the performance by removing one layer from hardware to the virtual machine [43]. The virtual machines is unaware of the virtualized environment and acts like it is running directly from hardware. This makes it possible to install most operating system because it does not require any modifications.

Para-virtualization

Para-virtualization provides the best performance by modifying the guest operating system so it becomes aware that it is a virtual machine which makes it possible to interact with the hypervisor [43]. The drawback is that it demands certain changes made to the kernel level and is therefore mostly used by open source operating systems like Linux.

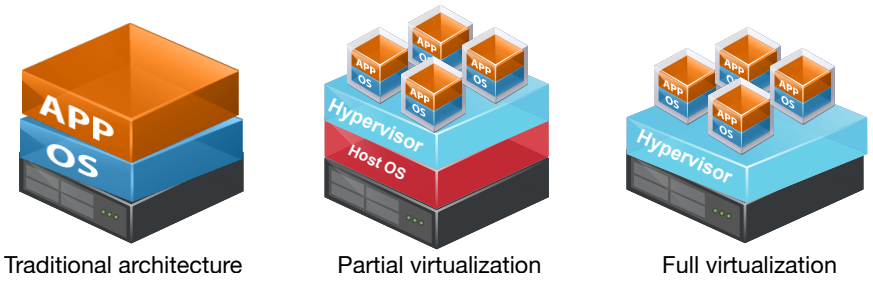

Figure 2.1: *Illustration of the three different virtualization techniques*

2.2 Hypervisors

A hypervisor is the layer between the physical hardware and the software layer, it is responsible for running the virtual machines and handle it translations from hardware to the virtual machine operating system. OpenStack iis the cloud computing platform and will be described later in this chapter. OpenStack requires a hypervisor to handle the hardware and the most popular choices are KVM, VMware and Citix XenServer. In OpenStack the default hypervisor is KVM and this might be obvious since OpenStack runs on Linux and KVM comes as standard for Linux distributions. The most important feature for the project is the ability to live migrate one virtual machine between multiple compute nodes, this is supported by all three.

2.3 Cloud computing

Cloud computing is a widely used term for a variety of computing concepts, and is commonly misinterpreted to be the same as virtualization. Virtualization is

one of the main technologies behind Cloud computing, but the difference is that cloud computing is the possibility to distribute computing over a network, while virtualization is the possibility to share computing resources between operating systems. Cloud computing have three major models known as software as a service, platform as a service and infrastructure as a service [32].

Software as a service

Software as a service (Saas) is a way to deliver software hosted by the cloud. This can be services as Email, CRM, virtual desktops and games among other. Saas has become a common way to deliver centralized business application either by the web browser or by using a thin client. Saas have been a key feature for companies to reduce IT costs by outsourcing hardware and software maintenance to a Saas provider. This is also one of the reason that the numbers of data centers have increased over these last years.

Platform as a service

Platform as a service (Paas) is like Saas hosted i the cloud but it provides a computing platform as a service. A platform can for instance be a database or a web server. The benefit with Paas is the possibility to offer a complex platform with service management as monitoring and workflow management without the cost of buying and managing hardware and software.

Infrastructure as a service

Infrastructure as a service (Iaas) provides machines and other resources like servers and load balancers. This alows costumers to deploy their own applications, and operating images in the infrastructure hosted on a data center or cloud provider.

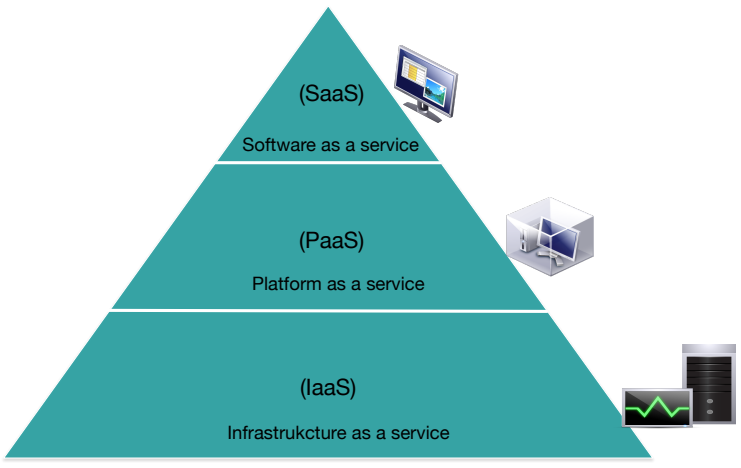

Figure 2.2: *Illustration of cloud services*

2.3.1 Data centers and Cloud providers

A data center is a facility used for computer equipment, where companies centralizes the IT operations and equipment. A data center houses critical systems and is therefor vital to run continuous without interrupts with focus on security and reliability. Data center architectures and requirements differs significantly from each other since the data center is customized after the services hosted. Some data centers host more web based applications that can run for instance in the browser, while other data centers are built for hosting classified data for governments and other security agencies.

The evolution in data centers has these last years been major due to the adoption of technologies like virtualization and cloud computing. Cloud computing became a paradigm for data centers providing for instance infrastructure as a service. This facilitated the possibility for the data center to rent out complicated infrastructures on-demand to customers without having all equipment in-house in the customers organization.
Over the past years the numbers of cloud computing providers have increased but the biggest growth has been inside the providers. By the increasing demand the providers have expanded their data centers and the largest cloud computing providers are [17]:

- Salesforce
- Amazon
- Google

These companies have hundreds of thousands of servers and several of these have multiple data centers. Google for instance had in 2009 more than 500 000 servers consuming electricity for more than 38 million dollar [31]. The physical equipment in data centers includes servers, storage and other types of hardware placed in racks for optimizing cooling and the organizing of cables. All the infrastructure placed in a data center should have a high level of availability, which means uninterrupted power source, controlled environment in form of cooling, air condition and physical security.

2.4 Cloud Platforms

Over the last decade several cloud platforms have occurred. Some are open source as Openstack and other are highly priced software solutions as VmWare.These are some of the Cloud software alternatives:

VMware [1] is one of the leading virtualization companies in the world for building cloud infrastructures. vSphere is widely used by over 250 000 customers and supports 2,500 application. VSphere have ESXi as virtualization layer, and features as auto deployment, network i/o controller, update manager among many other features [15].

Figure 2.3: *VMware logo*

Openstack [2] is a collaboration of developers for making a open sourced cloud computing platform for both private and public clouds. Openstack is a cloud operating system that is able of controlling large pools of compute, storage and network resources. All this is accessible through a dashboard that gives the administrators a good overview and control. Several large companies uses Openstack for their cloud, like HP [29].

Figure 2.4: *OpenStack logo*

Amazon Web Service(AWS)[3] dominates the public cloud zone by 8 geographical "regions". Amazon offer several services but one of the biggest is "Amazon Elastic Compute Cloud"(EC2), this service offers resizeable compute capacity in the cloud. One of many benefits is a design to make web-scale computing easier for all the developers. Along with EC2 amazon also offer "Amazon Simple Storage Service" (S3) which enable storage in the cloud while EC2 handles the computing. [38]

Figure 2.5: *Amazon logo*

[1] http://www.vmware.com/products/vsphere/
[2] http://www.openstack.org/
[3] http://aws.amazon.com/ec2/

2.5 OpenStack

OpenStack delivers massively scalable cloud operating system that is capable of controlling large pools of compute, storage and network resources. All these features can be handled from a web interface. OpenStack is widely used by small private users to some of the world's largest public providers like Hewlett-Packard and Rackspace. Rackspace is also one of the founders of OpenStack. One of features that have made OpenStack grow to be large is the compatibility with Amazon EC2.

2.5.1 Architecture of OpenStack

To be able to deliver the massively scalability OpenStack does, it is divided into three divisions; Openstack(Nova), OpenStack Networking(Neutron) and OpenStack Object/Block Storage(Swift/Cinder).

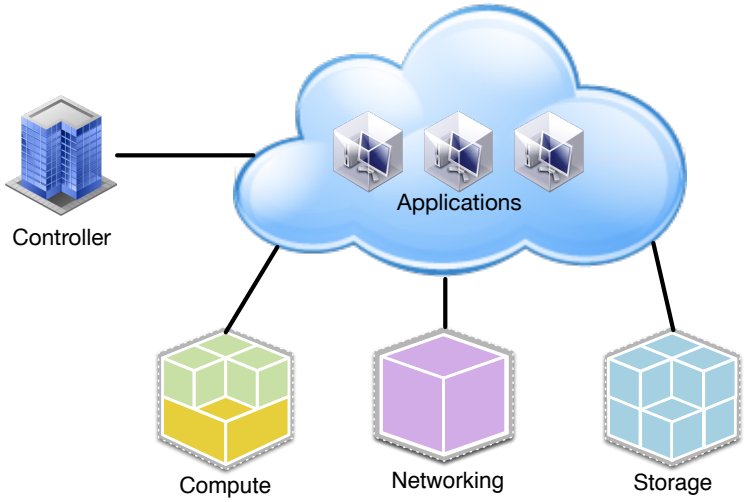

Figure 2.6: *OpenStack architecture*

2.5.2 Compute(Nova)

Nova is the code name for the compute nodes in OpenStack [13]. Nova can be deployed by using one of many supported hypervisors like XEN or KVM. Nova manages all the virtualized server resources like CPU, memory and Network interfaces. The administration of the virtual machines is handled by the compute node and can be live migrated between compute nodes to increase efficiency and flexibility.Virtual machines and images can be imported and shared while the machines can be rebooted, suspended, re-sized or terminated. The compute nodes keeps the virtual machines separately and secure. Through the web interface it is even possible to connect directly to the virtual machines. The connection and commands to the compute nodes can be done through application program interface(API) or directly to Nova by commands. In this thesis Nova has a

central role in form of controlling resources and handling live migrations of virtual machines.

2.5.3 Networking(Neutron)

The network in today‘s data center is more complex than ever before and a part of the complexity is the ever increasing number of machines and devices [39]. Neutron is a API-driven network that manag networks and IP addresses. These settings can be administrated from the web interface. One important feature is the VLAN‘s that by default separates the servers and traffic. All traffic can dynamically rerouted which is important in a scenario involving live migration.

2.5.4 Storage(Swift/Cinder)

OpenStack Storage provides object or block storage specialized for use with servers and other applications. Swift offers cloud storage with a simple API that makes it possible to simply retrieve lots of data. Swift is scalable and optimized for storing unstructured data. Cinder provides the managing of volumes in OpenStack [12]. This was in the beginning called nova-volume, because it is one of the key features for making live migration work.

2.5.5 OpenStack with KVM and libvirth

OpenStack is the software controlling all the virtual machines but the handling of the hardware is controlled by KVM and libvirth. Libvirt is provides a communication application programming interface to make the communication to KVM as easy as possible. Libvirt uses the command line utility "virsh" which makes it possible to send commands to manage all virtual machines.

2.6 Manage Large Networks - MLN

Manage large networks(MLN)[4] [1] is a management tool for building and administrating large groups of virtual machines, that can be used together with OpenStack, Xen, UML and together with Amazon EC2. MLN is written in Perl and is built with easy understandable configuration files to handle virtual machines as a project which makes it powerful easy to create a hole set of machines with one superclas that defines all the necessary information for all machines. This can be specified for each virtual machine as well but the script it cleaner with a superclas and just some corrections in each VM. A project can be built, started and stopped by simple commands.

[4] http://mln.sourceforge.net/

2.7 Openstack Database

OpenStack and Nova stores all the information as compute nodes, instaces and users in a relation database. The upside of this databse is the possibility to query the database for the exact information wanted. This is the query and tables for the instances and compute nodes.

The query for the database is select all from instance\G, the \G is for presenting the output in orderly manner.

Compute id	Unique Id
service_id	Service reference Id
vcpus	num of vcpu
memory_mb	memory size by mb
local_gb	local disk size by gb
vcpus_used	num of used vcpu
memory_mb_used	size of used memory by mb
cpu_info	cpu details info represented as json format

The query for the instance database is select all from computenodes\G

id	Unique Id
name	instance name
user_id	instance owner user
Compute	Located at compute node
VCPU	Number of virtual CPU

The response from the database contains all information from the compute nodes if more detailes are not specified. This is an example of some of the response.

OpenStack database response

```
1 hypervisor_hostname: compute10
2 service_id: 16
3 vcpus: 64
4 memory_mb: 257938
5 vcpus_used: 6
6 memory_mb_used: 9728
7 hypervisor_type: QEMU
8 running_vms: 4
```

2.8 Energy efficiency in data centers

Energy expenses is a significant part of a data centers operational cost and the physical racks filled with servers and the cooling of these racks is the big consumers of electricity. Million of dollars are spent powering these racks and according to the ongoing trend the data centers are growing fast and so do the electricity price. According to the usage Google had in 2009, a small reduction of only 3 % would

reduce the cost by over a million dollar. [31] [10]

2.8.1 Live migration of virtual machines

Live migration is a term used to describe the process of moving a running virtual machine with any operating system from one physical host to another. These hosts can be in an OpenStack scenario be from compute node 1 to compute node 2 without any downtime for the virtual machine.

Live migration offers flexibility and improved performance to the infrastructure, but it has certain requirements. The virtual machines migrated to another hosts need to have a shared storage that can be accessed from both hosts. The storage can not be live migrated so the system requires a separate storage. Not all hypervisors today supports live migration in OpenStack and therefor the choosing of hypervisor in this project is important. Some of the hypervisors that do suport live migration is KVM, QEMU, XENServer and HyperV [16] [28].

A live migration can happen in three phases:

- Warm-up phase
- Stop-and-copy phase
- Post-copy phase

The names of the phases are self explaining by describing when the memory used by the virtual machine should be moved from the host to the destination. The warm-up phase sends all memory from host to the destination before the actual move of the virtual machine happens. Stop-and-copy stops the machines and then moves the machine while post-copy moves the virtual machine before starting to send the memory [42].

Cost of live migration

The cost of live migrations are two-folded. The prize to pay is both performance vice at virtual machine and the and the extra electricity consumed. Several studies have tried to model the behavior and Voorsluys et al. described the degradation including the downtime to be 10% for a web application [37]. This will vary from application to application. The degradation is minor but the number of live migrations should still be minimized. The length of the migration is also something to consider. Since the migration mainly is moving of memory there will take longer time depending on memory size and bandwidth. Beloglazov et al. described the performance degradation experienced by the virtual machine in the article Adaptive Threshold-Based Approach for Energy-Efficient Consolidation of Virtual Machines in Cloud Data Centers to be like (2.1) and (2.2) [2].

$$U_dj = 0.1 \cdot \int_{t_0}^{t_0+T_{mj}} U_j(t)\,\mathrm{d}t \tag{2.1}$$

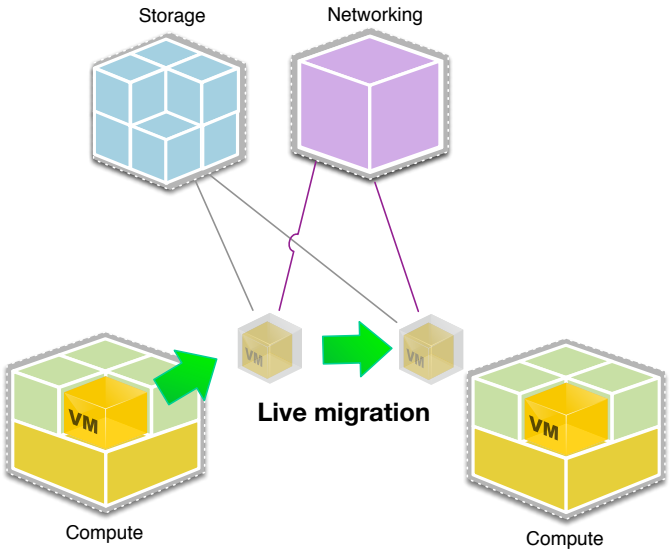

Figure 2.7: *Live migration*

$$T_mj = \frac{M_j}{B_j} \tag{2.2}$$

In this equation the U_{dj} is the degradation of total performance by the virtual machine, the start of the migration is t_0 while T_{mj} is the totalt time of the migration. The interesting with this equation is the $u_j(t)$ which is the CPU utilization by the virtual machine. M_j is the memory used the virtual machine and B_j is the available network bandwidth [2]. One this not included in this equation is the size of the packaged sent over the network. If supported it might be possible to use a jumbo frame which supports more than 1500 bytes of payload. Jombo frames can take up to 9000 bytes which could lower live migration time [27].

2.9 Dynamic scaling in cloud computing

One of the major advantages in cloud computing is the ability to scale and customers have the opportunities to scale in a virtually infinite computing infrastructure [35]. Along with the possibility to scale comes the advance billing mechanisms allowing customers to use a pay-per-use model. The benefits are obvious as customers now only pay for the actual use and not for the idle hardware. This have created a demand for services that can adjust the number of machines by scaling up or down after the actual load of the system. By scaling resources according to the usage the operational cost of the system will be adjusted according to the usage which can possibly save a lot of operational cost for the providers.

The growth of data centers have become a larger environmental problem and have therefor also provoked a great deal of research and development. The research can roughly be divided into two main groups: Dynamic voltage/frequency management inside a server and scaling of the virtual environment.

2.9.1 Relevant research in power-aware scaling

Power management in large-scale data centers is necessary to both address the operation cost and to make the required cooling as low as possible. This has provoked a massive amount of research to find different approaches for reducing the power consumption without creating disadvantages in form of performance and manual tasks that needs to be done by the administrator.

VirtualPower
The VirtualPower [26] project explores the possibility to integrate online power management mechanisms and policies in a virtualized environment. The goal of this project is to efficiently manage workloads through migration technology to increase power efficiency. The implementation was done in a Xen hypervisor. Nathuji et. al [26] was able through VirtualPower management to improve the power consumptions by 34 percentage in heterogeneous server systems.

Vgreen
Dhiman et.al [6] uses Vgreen as a multi-tier software tool to make virtual machine management in a clustered virtualized environment more energy efficient. The project uses different metrics to capture power and performance characteristics for both virtual and physical machines and uses these characteristics to implement policies for scheduling and power management. The system was implemented on a Xen hypervisor and improved both average performance and system-level energy savings by 40 percentage. This is interesting because it is possible to see that dynamic voltage and frequency scaling can save a considerable amount of energy and highlights the difference between shutting down a server and just scaling the amount of electricity the server is allowed to use when it is in an idle state.

DVFS
This project focuses on the reduction of electricity consumption through dynamic voltage frequency scheduling (DVFS) [36]. It is done by calculating the resource usage from virtual machines in a cluster, before the algorithm dynamically scales the supplied voltages to the server. By lowering the voltages, the performance decreases which makes it important to know how much hardware that is required to handle the load.

2.9.2 Service level scaling

After the cloud providers started to offer a more advance billing model with the possibility to pay-per-use the research for dynamically scaling software and services drastically increased. The results of the research and development was elastic software and services with the possibility to acquire and release resources in response to dynamic workload. The benefits are obvious, one only pay for what that has been used. Below follows three interesting projects that scale after the actual workload.

2.9.3 Relevant research in performance application scaling

Hadoop
Hadoop [19] is a project that addresses an elastic control for multi-tier application service that allocates and releases resources as virtual server instances. The main focus was to add and remove storage nodes. This have been tried before but this projects focuses on decreasing the delay when the storage nodes needs to rebalance after one node has been removed. A new controller was designed and implemented to show how the controller(the Hadoop distributed file system) adapts to workload changes to maintain performance objectives efficiently in a pay-per-use model.

The OpenNebula engine
The OpenNebula [25] project uses a virtualization layer between the service and the physical infrastructure. This means that on top of the infrastructure provided by Amazon EC2 they have implemented a virtual machine manager called the OpenNebula engine. With the flexibility that OpenNebula provides, the project separates the resource provisioning from the service management. This gives benefits as dynamically adapting to the workload. Two different cluster-based services were implemented and analyzed, showing a sustained performance increment.

Autoscale
Gandhi et. al. [46] introduce a dynamic capacity management policy, Autoscale. By adding or removing servers Autoscale greatly reduces the number of servers needed, while still meeting response time and performance requirements. Autoscale handles applications or webservers and this is done by using two key features: maintaining the right amount of server spare to handle sudden increase of demand, and the request size and efficiency. One thing that differs Autoscale from other approaches is that the software does not try to predict the future request rate, but it introduces a smart policy that always should have resources to handle a sudden changes in request rate. The result of the project a significantly lower response time and the power consumption.

2.9.4 Hardware scaling

When power-aware scaling and service level scaling is combined the result is software that is able to both scale the service and after scaling the software should decrease the power consumption by either lowering the voltage or turning servers completely of. The projects that follows below have tried different approaches for adjusting hardware according to the workload. These projects are highly relevant for this thesis since they have scaled hardware after workload.

MPC
Zhang et.al [47] used the Model Predictive Control(MPC) to find the optimal control policy for dynamic capacity provisioning. A system that controls the number of active servers in a data center for energy savings. The solution aims to find a trade-off between energy savings and capacity reconfiguration cost. The framework is an initial step towards building a full-fledged management system.

RSOM
RSOM [5] stands for Recurrent Self-Organizing Map and was a module in a project to make an energy-efficient self-provisioning approach for cloud resource management. The project was based on an unsupervised predictor model in the form of an self-organizing map that predicted the user load after historical usage. The main unique feature is the ability to save energy by resource administration strategy. The aim of the project is to make a system that can adapt itself to meet requirements of performance, reliability and security without manual intervention, by self-configuration, self-optimization, self-healing and self-protection. The project did show that the proposed approach has provided promising results by using the Cloudsim framework which is a simulation tool for clouds.

Green scheduler
Another interesting project is done by Duy et. al. [7]. The project used four algorithms: prediction, ON/OFF, task scheduling and one evaluating algorithm to build a green scheduler for reducing energy consumption. They designed, implemented and evaluated the green scheduler for reducing energy consumption of data centers in a cloud computing simulation environment. The project used a neural predictor algorithm to predict the usage which the ON/OFF algorithm used to find the right number of servers to handle the load. For the evaluation the project used Cloudsim and Gridsim to simulate both the cloud and the usage. The conclusion and result of the project is 49,8 percentage reduction of energy consumption in the simulation. As future work the project suggests to improve and building a model which can actually be used in real data centers.

2.10 Similarities in the approach

All three projects have different approaches but with a closer look at the algorithms the three project are quite simular. MPC have a model where the system is built with five modules which is:

- The schedular
- The monitoring module
- The prediction module
- The controller
- The capacity provisioning

These five modules makes the system able to find adjust the system after the usage or workload. In comparison the Green scheduler project have four algorithms doing the same job. These algorithms is divided into four groups:

- The prediction algorithm
- The ON/OFF algorithm
- The schedular algorithm

- The evaluation algorithm

The last project RSOM is divided into four main groups which is:

- User‘s need
- Self-control policy
- Knowledge data base
- Autonomic provisioning model

The three models have one module responsible for the assigning incoming tasks or workload to active machines in the cluster and to log the traffic for analysis done by another module or algorithm. One module is monitoring the resources to find the best amount of resources and reports to another process that receives all the statistics to make some kind of predictions for the future usage. MPC and RSOM project have a module responsible as a controller either named as controller or more complicated explained as a autonomic provisioning model with a self-administration module. The Green scheduler project has a self explaining ON/OFF algorithm doing the same thing. One difference between the three projects is the learning process. The Green scheduler has an evaluating algorithm with the ability to avoid over-reacting to noise in workload fluctuations. An interesting approach is proposed by two training policies for the algorithm. Either static training or dynamic training. Briefly explained the static training policy takes place at regular time intervals to find persistent load increases and decreases like when users tend to occur. This evaluation can be done hourly or daily. One downside is that it does not take performance into account. The dynamic training on the other hand maintains an error term "Mean Squared Prediction Error" (MSPE) [7] which observe the threshold of error. When the system is stable there is nothing triggered but if the error threshold is triggered by this algorithm:

$$MSPE = \frac{1}{S1} \sum_{i=T-S1}^{T-1} (P_i - A_i)^2 \tag{2.3}$$

T is the current time, S 1 is error threshold, while P_i and A_i is the predicted workload at time i. The evaluating algorithm is able to trig an adjustment of running servers but the project also explains that predictions error are unavoidable.

The RSOM project has chosen a simpler prediction model with a sensor, actor and effector. The module predicts loads in the future based on the historical load simular to the static traing phase in the Green scheduler project. The last project uses a prediction algorithm called "Auto-regressive Integrated Moving Average" model, ARIMA [4] . The model is used to predict the time series G_K^T where r is type of resource and time in this case is k.

$$G_{k+1} = \phi_i L^i G_K + \varepsilon_{k+1} + (\sum_{i=0}^{q-1} \Phi_i L^i)\varepsilon_k \tag{2.4}$$

The element $G_K^T + 1$ is the prediction made by the algorithm, it states what that will come next. The term ε_k is simply the error term assumed to be independent. $L^i G_K$ is the past element, and can also be written as

$$L^i G_K = G_K^T - 1 \tag{2.5}$$

The values n and q is the last measured values.

In many ways is the algorithm used in RSOM a more advance and complex algorithm combining both MPC and Green scheduler modules.

2.10.1 Energy consumption and CPU usage

In the pursuit of the most efficient usage of electricity it is important to understand the relationship between the CPU utilization and power consumption. The power consumption of a physical machine has in research been estimated as a linear function of CPU, memory and disk usage [47]. This even applies when running with dynamic voltage and frequency scaling [3]. Throung Duy et.al proved this by gathering empirical measurements of two machines over time while running load generators at certain levels. [7]. Moreover, the most interesting is in fact that studies show that an average idle server consume 70 percentage of the power consumed when it is fully utilized [3]. The power consumption is expressed by Anton Beloglazov et. al. as a function of the CPU utilization (P(u)) [2].

$$P(u) = k \cdot P_{max} + (1 - k) \cdot P_{max} \cdot u = P_{max} \cdot (0,70 + u) \tag{2.6}$$

P_{max} is the consumption of the server, while the consumption of an idle server is k. The CPU utilization (u) change over time which makes it a function of time u(t). Anton Beloglazov et. al. defined the total energy consumption by a server as [2]

$$E = \int_t P(u(t))dt \tag{2.7}$$

2.11 Bin packing

All the projects above are in one way or the other trying to pack all virtual machines on as few physical machines as possible. This is known as the mathematical concept "Bin packing" [40]. The essential feature of Bin packing is to find a way to pack a finite number of objects in different sizes, in as few bins as possible. The approach to the packing is many, such as 2D or 3D packing, linear packing or packing by a weight and so on. One of the fascinating facts of bin packing is that there is always perfect solution but this can in many cases not be ideal because it might take a too long time to calculate all possible combinations.

One example of bin packing can be a moving company with one truck and 10 boxes. Figure 2.3 shows the different sizes of boxes compared with the truck.

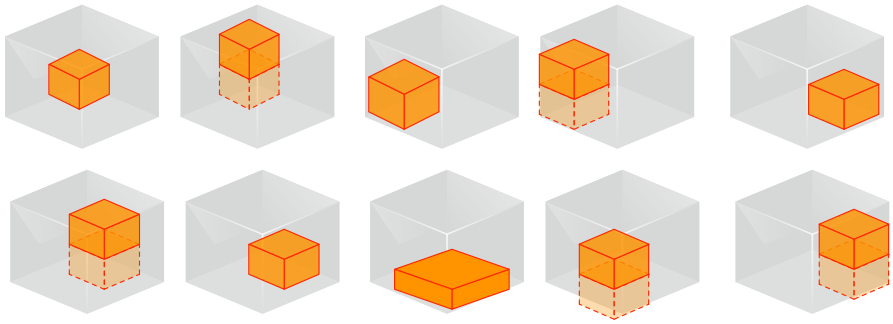

Figure 2.8: *Boxes for bin packing*

Bin packing is how to best pack all these boxes into one single bin.

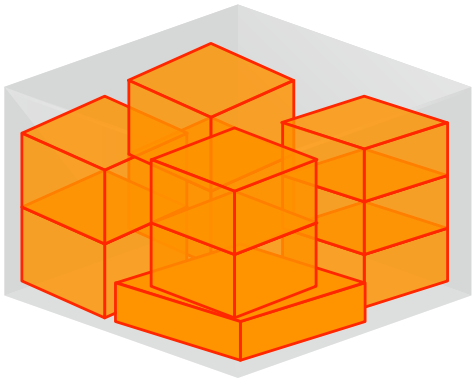

Figure 2.9: *A completed bin packing*

In bin packing there are some problems that are categorized as an Non-deterministic Polynomial-time hard problems (NP-hard problem) [22]. In bin packing the number of items to pack gives an exponential time used to find a solution. This means that some NP-hard problems can be so complex that it can not be

solved by a standard computer in polynomial time. This is a mathematical problem not yet solved, so the solution is to find a close-to-optimized solution that is not perfect but the best solution calculated by certain time.

The complexity in bin packing for an environment with virtual machines is NP-hard problem because of all the variables constantly changing. When calculating the optimized bin packing solution for the system the state is changed before all possible combinations are tested. This means that the calculation would have to calculate the bin packing on certain variables and not all. These can be state of system(idle or working), memory and virtual CPU.

The example in figure 2.4 and 2.5 is bin packing in 3D. Formally the "V" is used for the value of the objects and bins "S" while "n" is the number of items. The number of bins is referred to as B.

First fit bin packing

The first fit bin packing concept is a straightforward robust algorithm. This kind of bin packing is where the algorithm for each item, tries to place the item at the first bin with enough space. If there is no room for the item the algorithm will open a new bin [23].

First fit- decreasing bin packing

This algorithm is a more advance algorithm than the first fit algorithm. However, the concept is the same, but when the algorithm selects an item it first tries to place the largest item. By taking the biggest item first it can optimize the packing more than with first first algorithms. This might be a way to minimize the live migrations [41].

2.12 Relevant work in bin packing projects

Bin packing has been a interesting problem for mathematics for many years but bin packing virtual machines and servers has occurred along with live migration possibilities in the cloud. The ability to relocate servers without shutting down the machines provides an additional opportunity not only to scale the service but also adjust the power consumption. The primary goal of the packing of virtual machines is to utilization of the space within in all physical servers to minimize the number of running machines. There are several contributions to the research for the bin packing in both 2 dimensional and 3 dimensions. Below is some of the relevant work.

2.12.1 BtrPlace

Btrplace [11] is a flexible consolidation manager for highly available applications. This is a manager for resources in data centers which dynamically consolidate workloads. Btrplace provides administrators the ability to give virtual machines

placement constrains and requirements to gain the best performance. Btrplace is a good example of how modern bin packing algorithms with constraints can be used to perform changes to a cloud environment. Btrplace was tested on a simulated datacenter containing 5000 servers hosting 30,000 VMs, and it could find a viable configuration in under 3 minutes.

2.12.2 Space Defragmentation Heuristic for 2D and 3D Bin Packing Problems

This study illustrates a technique to make the packing process more efficient by combining small unused gaps in bins or containers to make room for more objects in each container [48]. The project presents both a 2 dimensional and 3 dimensional bin packing algorithms that outperforms the other meta-heuristic approaches as the algorithms proposes a defragmentation technique that is closer to the human way of packing items into containers by replacing small items with bigger ones and to make room for more items. These are factors that need to be taken into consideration when making an algorithm for packing virtual machines.

2.12.3 Dependable Virtual Machine Allocation

This project [45] proposes a solution where the usage of CPU is measured to find an optimal match for VMs that can be located at the same physical machine. This can be compared with a company that has one company car divided by people which work at shift, which means that there is no need for two cars since the first one is finished when the second worker comes to work. In the same way is this done to virtual machines where the relocation mechanisms finds complementary machines to work at the same physical machine to minimize the number of required servers or to maximize the performance.

2.13 Tools for modeling and implementation

As described in the background chapter the environment will be built in an OpenStack environment with KVM as a hypervisor. These tools will be describes below to get a clear overview of what each tool will do.

2.13.1 Dell Remote Access Controller - DRAC

DRAC is an interface card in the server, that provides out-of-band management features like live power consumption. The interface has its own processor, memory and network connection and can be explained as a little machine inside the server that can give remote access to the server as if one was physically presence at the server. The two main function used in the CERES project is the ability to monitor the actual power consumption and starting and stopping servers remotely. This makes it possible to clearly see the effect the policy has on the environment.

2.13.2 Perl scripts

To build a working prototype an essential part is thoroughly understanding of all technologies, and the overview of the possibilities. Perl is the scripting language used for building the glue between all these different programs that is required to work together.

2.13.3 Storing the data - OpenTSDB

To store result and measurements in the environment OpenTSDB will be used. OpenTSDB is a database used for time series [14]. This means that is it collects sets of datapoints over time, like retrieving information about power consumption for all hardware every minute. These sets of data can later be shown in OpenTSDB data plotting system, which both can draw graph or give the graph in numbers for later analysis. This allows the project to continuously store data from the start to the end and the ability to go back and retrieve any data form any given time.

2.13.4 Sensu

Sensu is used for keeping track and monitoring of the environment. Sensu is a open source monitoring framework. Sensu provides the ability to meet our special case and requirements. Sensu is created to meet new monitoring challenges like the CERES project and other cloud environments.

2.13.5 Pseudocode

Pseudocode is a high-level description which is written in the same style as if it was written for a operating system. Instead of using syntax for a computer the script in pseudocode is written in a way that is easy to read and to understand for any human and not only for those how can write code. This will help to get an explanations of the new features or new approaches. Pseudocode will in this case be used to complement diagrams and figures.

This is an example of how pseudocode can be written:

Pseudocode of shuting down a physical machine

```
1 IF all VM's is removed form physical server
2     THEN
3         Shutdown server
4     End
```

This code describes a process where all virtual machines is moved out of one server is therefor asked to shutdown. After the shutdown the physical machine waits for a new command from controller to boot up to receive virtual machines is required by the prototype.

2.13.6 Basic Representation of Interactive Components - BRIC

The purpose of Basic Representation of Interactive Components (BRIC) [8] is to give an illustrative description of functions to all agents and components in an

architecture. BRIC has a component approach to a multi-agent system which can draw associations to electronic circuits. Just like electronic circuits the BRIC models are connected between components with lines or wires sending messages from output terminals and components receives information from input terminals as illustrated in figure 2.9.

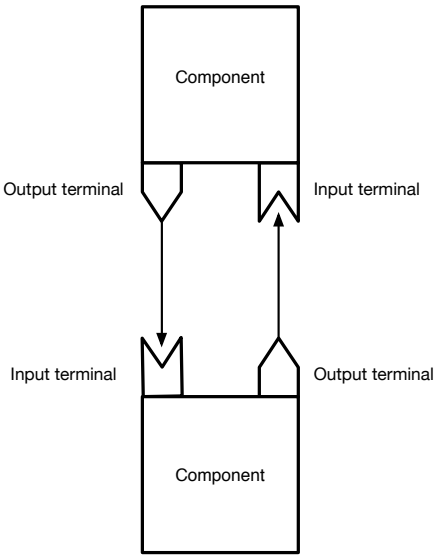

Figure 2.10: *BRIC modeling*

BRIC will be used in the design of the prototype to demonstrate the work flow. BRIC is built to describe a multi-agent system, while the prototype will be a distributed system. This means that the BRIC model will be adapted to illustrate the process of the prototype.

Chapter 3

Approach

This chapter will describe and explain which actions that will take place to best answer the problem statement: **How can energy efficiency be optimized by dynamic scaling of the compute nodes in a cloud environment?** The problem statement describes several key features like: building of a working prototype for dynamical scaling of compute nodes, and how different policies can optimize the energy efficiency.

The approach will explain these different aspects:

- Elements in the environment
- Features of a policy
- Design of model
- Implementation of model
- Experiments and scenarios with different policies
- Expected results

3.1 Objectives

The objectives in the project will be described in this chapter by placing terms and concepts from the background chapter into a clear structure to make an autonomic prototype. The practical part of this project is essential and the approach can be divided into three properties. The first phase is to design models for the prototype including three algorithms and prerequisites. Phase two is to implement these for experiments. The last phase is to do measurements and to do analysis of these measurements to support claims made in the introduction.

1. Design of an autonomic prototype
    (a) Create a model for the prototype for dynamic scaling of the compute nodes in a cloud environment.

(b) Create three models of algorithms to bin packing of virtual machines with formal definitions, pseudocode and illustrations.

(c) Identifying tools, prerequisites and features for the prototype

(d) Hardware and architecture overview

2. Implementation for experiments

(a) Implementing monitoring and collection of empirical evidence

(b) Building the system by following models form phase one.

(c) Implementing three algorithms for experiments.

3. Measurements and analysis

(a) Execute tasks to test prerequisites and the environment

(b) Perform experiments for algorithm 1,2,3 to prove the concept

(c) Execute simulations to find the best algorithm for optimizing the power consumption

(d) Long term test of the best algorithm

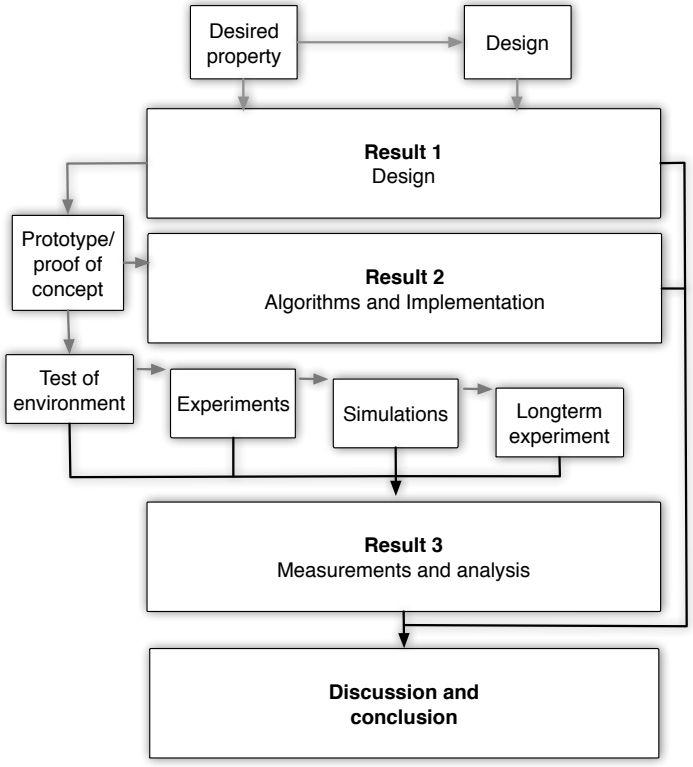

Figure 3.1: *The approach*

These three phases all requires deep understanding of the technologies described in the background chapter.

3.2 Understanding the complexity of a working framework

Cloud computing is already the most complex system available today and how can the most complex system be more efficient without making the system unusable for system administrators? A difficult task will be to find a balance between not adding more work and manual tasks to already overworked system and network administrators but still make a system that saves energy by scaling without hurting the performance.

There are four axis of difficulty in this project for getting a working prototype. The first axis is the technical difficulty which is the main difference between a simulation and a actual working prototype. The prototype will be tested in a real environment and there are technical details which can sound like a simple fix which in reality can take several days to fix. This prototype will not only control one machine but 11 compute nodes with shared storage and network. This requires a complex model which can handle a large amount of constraints and adjustments. A live migration is the ability to move one virtual machine while it is still running without any interruption. This means that storage and network always has to be available. The second axis is therefor the complexity required by the model. To be able to document the effect of the prototype there is a need for evidence, which will be in form of power consumption and performance of the environment. This data has to be accurate and precise for the analysis. The large amount of data makes the third axis of difficulty which has to be solved by a well established and robust database.

The last axis of difficulty in this project is the experiments. Since the experiments will be done in an environment in production, the experiments have to be well planned and tested to not cause any damage or inconvenience for other active users of the cloud. The prototype will make significant changes to the numbers of active compute nodes and move a large amount of virtual machines. For this prototype to be a success this have to done without any inconvenience for the users and administrators.

3.3 Design phase

The design phase will be a combination of developing models to scale of compute nodes in a cloud environment and developing models for three algorithms able to move virtual machines to optimize the power consumption. To make this possible there are several features required by the environment, and these prerequisites will identified. These might inflict the architecture and design of the hardware architecture.

As shown in figure 3.2 there are several layers of operators both in terms of hardware and OpenStack controllers like Nova and KVM. From the bottom there are physical hardware controlled by KVM, and the OpenStack cloud is built on multiple nodes that work together to host all virtual machines hereby called VM. Both

hardware and VM are controlled by the OpenStack layer which is a hypervisor.

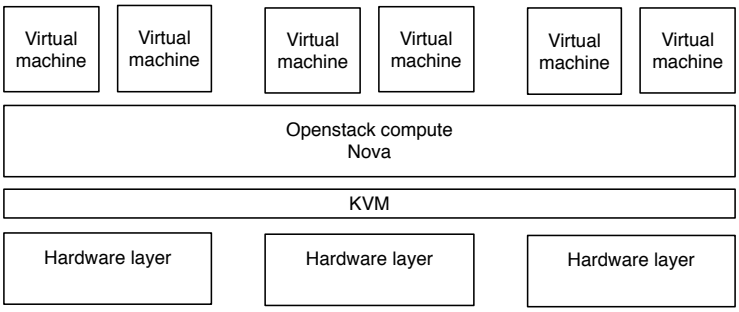

Figure 3.2: *Design phase*

All theses layers have unique features and responsibilities which is of interest to measure:

1. Hardware layer
   (a) Power consumption
2. OpenStack layer
   (a) Active compute nodes
   (b) Number of virtual machines per compute node
3. Virtual machines
   (a) Resources used
   (b) Importance

The terminology that describes all physical machines with associated VM‘s, will be refered to as a *set*. It is necessary to set a terminology that describes the compute nodes, virtual machines, which virtual machines thats is running on which hosts, the usage of CPU and memory, thresholds and utilization.

3.3.1 Modeling

The complexity of the environment and the desirable characteristics of the prototype makes it important to divide the design into several models. This to get an overview of all features, functionality and tasks. The models will be both described in text, pseudocode and graphically displayed to help the reader to easily understand the prototype without needing read the actual code.

The design phase will first describe one model for the practical parts of the prototype followed by three models of algorithms to best answer is the dynamic scaling of the compute nodes. It is important that the algorithms can make an informed decision regarding changing the state of the system. This will include observing

and monitoring the system to see whether there is a need for more or fewer servers. This model is required to both see the capacity in terms of performance and power consumption. The model will have to make certain rules for what to look for and how to react.

Furthermore actions are required for the actual scaling of the physical machines. This will be triggered by the algorithms. The models need to describe the actions of

- Observe the environment
- Decide if a change is required
- Determine where to move virtual machines
- Move virtual machines
- Confirm the success of the migrations
- Confirm the new state of the system

3.3.2 Policies

A policy is a protocol to make decisions to achieve a outcome. In this case the prototype will prove the concept but will not be a policy before the algorithm is applied. The algorithms together with the prototype will make three policies. Policy one will have algorithm one as its decision maker and policy two will have algorithm two. The last and third policy will use algorithm 3.
The first policy is a basic policy which will demonstrate that the mechanisms built in the prototype are working according to the model and that the features are working properly. One of the main properties to this policy is to test and experiment with the robustness and stability to the prototype.

Furthermore, there will be implemented one policy that is more advance. This policy will take more consideration of the different aspects of performance and the number of live migrations.

The third will be a known three dimensional bin packing algorithm to compare the two algorithms developed in this project specified for migration of virtual machines.

3.4 Implementation

This project will be based on an environment built and managed at the University college of Oslo and Akershus. Hardware and bandwidth is provided by the college. Since this project aims to build a working prototype it is important that the objectives are measurable in form of efficiency and power savings. The prototype will be built by using the programming language Perl, and the implementation chapter will cover the essential parts of the scripts by showing the most important part. An interested reader is advised to look in appendixes for complete scripts.

When programming large and complex software like this each single module of the script has to be tested to make a robust script. The implementation of the scripts monitoring the environment will be presented first, then the implementation of the prototype will be examined.

3.4.1 Environment

To answer the problem statement there will be implemented an working prototype. This prototype will have the models described above as fundamental bricks and the policy will decide how the system will react. The number of policies will be delimited due to time limitations.

To gather empirical evidence for answering the problem statement the policies will be implemented in a real-life cloud environment that is in production. This is not a simulation but in fact a real software running in an real environment identical the systems running in the data centers all over the world.

3.4.2 Prototype

The prototype will be built with three separate parts which together makes the prototype to an autonomic dynamically scaling cloud. The three modules will be state, policy and action. State will be responsible for monitoring the system to provide the required information to the policy. The policy will evaluate the state of the system before running the algorithm to optimize the environment. New locations for the virtual machines needs to be stored and provided to the last module which will do the live migration of the virtual machines from one compute node to another.

3.4.3 Data collection

The purpose of the project is to provide data and evidence that the prototype can decrease the power consumption of a cloud in a way that has not been done before. This requires a robust and accurate gathering of data. This will be done by constant gathering data to the OpenTSD database. This database will contain information retrieved from the environment.

1. Compute names
2. Power consumption
3. Free VCPU at compute node
4. Free memory at compute node
5. Total number of running virtual machines
6. The number of running virtual machines at each compute node
7. The number of running compute nodes

3.5 Experiments

The experiments that will be conducted will be divided into three categories. The first category is called tasks and will consider all perquisites required to be working for the prototype to function as planned. When all these functions are working there will be experiments for all three policies. These experiments will contribute with the stability and robustness of the prototype as well as an indication of how successful the algorithm is. The comparison of the algorithm will first happen in the section where all policies are simulated in the same environment to measure the performance.

3.5.1 Tasks for testing the environment

Before any experiments of the policies could be done there where several key features required to work. This is a list of the experiments performed to ensure this prototype could be built. Task for testing the environment:

- Task T_1 Live migrate a virtual machine
- Task T_2 Check connection to databases and information regarding state of environment
- Task T_3 Check monitoring and data collection
- Task T_4 Test sending shutdown command to a compute node
- Task T_5 Test starting a compute node from IDRAC 6 interface
- Task T_6 Calculate the time a start up for a compute node

3.5.2 Experiments for policies

When all perquisites is in place there will be conducted experiments for all policies. These experiments is intended to give a proof of the concept and illustrate the different aspects of the three algorithms. All three algorithms will have different constraints which can make significant changes in the environment in terms of performance and power consumption. All experiments will first be simulated to see if there are some errors in either the calculations from the algorithm or the action required to scale the cloud.

Policy 1

Since the experiment is conducted in a production environment which is used by students and staff, all experiments needs to be simulated before running the actual policy which live migrates the virtual machines. Policy 1 is the simplest algorithm and therefore conducted first. These are the experiments:

- E_1 Simulate test run of algorithm and calculations
- E_2 Execute policy 1

Policy 2

Policy 2 is a more complex and will have more experiments and simulations. The policy will have additional constraints which needs to be tested before implemented in the algorithm. These are the experiment for policy 2:

- E_3 Simulate CPU affinity by script for all machines located at a single compute node
- E_4 Simulate to check constraints regarding number of live migrations and sorting of compute nodes
- E_5 Simulate a execution for proof of concept for the algorithm and calculations
- E_6 Execute policy 2

Policy 3

The third policy will contain a bin packing algorithm not explicitly created for moving virtual machines which can lead to a change in the experiment according to the functionalities of the algorithm. These are the preliminary experiments:

- E_7 Collect information and do calculations to input in algorithm
- E_8 Simulate a execution for proof of concept for the algorithm and calculations
- E_9 Calculate the number of live migrations and ensure the correct response

3.6 Comparison of the algorithms

After all experiments is completed the simulation for optimization will begin. This will compare the three algorithms and determine the difference between them and to see how the policies affect the environment. These simulations will be done in the exact same environment for a better analysis. The conditions and variance between the algorithms will be decided after the implementation and experiments. These simulation will be analyzed to find a suited policy for running long term testing.

3.7 Long term testing

One policy will be chosen for a long term test over three weeks. The purpose of this test is to see how the policy handles the environment over time. This will also identify how the policy handles bursts of virtual machines and performance constraints.

3.8 Expected results

Each policy with associated experiments will give an indication of how successful the prototype is optimizing the energy consumption. The empirical evidence from each experiment will be gathered from the database and analyzed.

The prototype will be tested in a running and real life environments and the expected results will be in form of saved energy consumption without compromising the performance. The power consumption will be logged directly from the physical hardware but can also be simulated by knowing how much the physical machines are using with different levels of CPU and memory usage. Since this is a highly complex prototype there is also expected to be several time demanding challenges, like live migration failure, script failure and logical challenges in different aspect of the prototype.

3.9 Appraising properties

The problem statement and introduction clearly describes several problems in today‘s data centers, and to measure and determine if this project and prototype is have achieved an improvement there is a need for certain clarification of how this can be measured.

The last phase is the evaluation of the implementation and prototype. The first two experiment phases which is tasks and experiments will be an exploratory study with focus in the functional aspects like showing that the prototype is working. The simulation part of the experiments will be an comparative study comparing the results from all three policies. As described below there are several factors that need to be identified as a feature, and monitored to see how the different policies can impact the consummation of electricity.

The term "energy efficiency" is used in the problem statement, and this can be measured in multiple ways. First of all, the term is the amount of energy a data center is using, and how much of this energy that is for something productive. Power Usage Effectiveness (PUE) is a calculation of how much of the total energy that goes to the computing equipment and not to other data center facilities like idle servers, cooling and lighting. Another way to determine how effective the energy is used is "Data center infrastructure efficiency (DCIE)" is a measure for the infrastructure in comparison of total amount of energy consumed by the data center. Both of these techniques for measuring the effectiveness is well tested ways to calculate the usage by the industry. The essence of these to techniques is to measure how much power the servers consume in comparison to what is gained form hardware. The energy efficiency will in this study be compared by measuring the amount of consumed energy by the original state and comparing this with the energy consumed by the environment after the policy has been conducted.

Optimize is used to describe the improving of the program or infrastructure to make

it run faster, perform better and to take full advantage of the resources that is available like time, cost, storage capacity. This can in be monitored in the infrastructure by multiple sensors reporting to the controller, and the difference between an environment running the prototype and the original setup will indicate if the problem statement is answered.

3.9.1 Autonomic Computing

Autonomic computing is a term referring to a distributed computing resource ability to self-adapt to unpredictable changes. This is one of the main features of the CERES project. The prototype is required to be self-managing. The reason that autonomic computing is important in this project is the fact that there is to many variables constant required checking. The conditions are under constantly changing which would be to hard to organize and keep track of for humans. It is just to complex. The model is required to be self-regulating according to changing components. This requires sensors for both self-monitoring, self-adjustments and a total awareness in the environment. The goal of the CERES project is to show how the prototype can contribute to self-optimization in the environment by autonomic monitoring and control of all compute nodes to ensure optimized conditions for energy efficiency.

3.10 Measurements and Analysis

The data gathered from the environment and inserted into the database is all collected with a time stamp which makes it easy to back in time and collecting all information regarding all aspects of the environment. One element not collected is the location of each single virtual machines. This is necessary in case some of the experiment would required a reset of the environment. This log will be collected before every experiment.

The experiments will generate a significant amount of data to analyze and the methodology of the analysis will be to only collect information between time intervals. All information can be retrieved from the database in text format which can be sorted and processed by scripts. These numbers will then be inserted into a diagram or figure to ease the readability.

The data to look for during the analysis will be the number of live migrations, amount of free VCPU, and number of virtual machines per compute node, number of running physical machines and the percent of reduction in power consumption. The graphs can make it easy to spot irregularities which can be hard to find by only looking at the numbers. The graphs will give a clear overview while the deeper analysis of the numbers can provide statistics. The purpose of this analysis will be to answer the questions from the problem statement. This is the preliminary approach and will be reconsidered throughout the project to find the best way to analyze the data. Similar research has had success with this approach.

Chapter 4

Result 1 - Design and models

In this chapter the design will be presented. This is a model of the prototype with the different modules, followed by three models of different algorithms. This chapter will also present hardware topology with different prerequisites, and tools. Last the experiments are presented. The CERES project is not only described and simulated but implemented in a working environment. This requires deep insight in all tools and technologies in the background chapter.

4.1 Model overview

Similar projects described in background show the different perspectives and approaches to creating greener clouds. However, there is little actual implementation into modern cloud frameworks. Three different bin packing algorithms will be presented that are subsequently implemented and tested in a production OpenStack environment. In this section we present the terminology used in the algorithm and present each in turn. The notations for the algorithms will be presented in the required section. This means that most of the notations will be listed in the presentation of the first algorithms and the next will only have additional notations required. In the next chapter, we discuss their implementations and results.

4.2 Model of the prototype

As earlier described the prototype will consist of three main modules. These three are state, policy and action. What each of these modules will do is described below. The main goal for the prototype is to first scale the cloud software wise before scaling the physical environment. The scaling of hardware requires that the physical machines are totally empty before it shuts down. If there is any virtual machines left on a machine going down the virtual machine will go down with the physical machine. The prototype will be built to find the state of the system before running an evaluation of the state. When the algorithm has decided what to do or not to do, the action module handles the changes suggested made in the environment. By designing the prototype like this it would be possible to change the algorithm without re-writing the entire prototype.

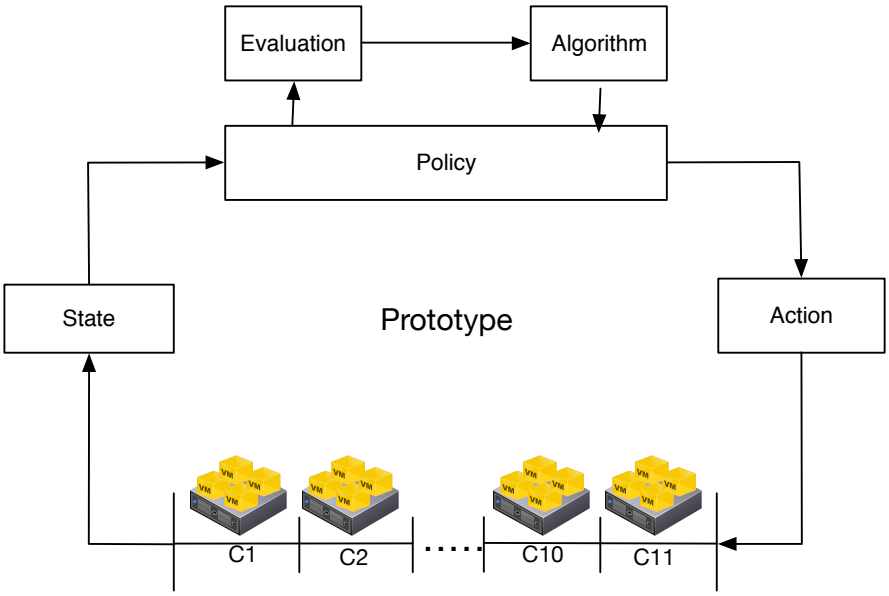

Figure 4.1: *A model of the prototype to illustrate process and modules*

4.2.1 The modules in the prototype

The prototype is built with three modules. These three will make an autonomic software able to scale the compute nodes in a cloud. This needs to be completely self managed system without manual tasks done by the system administrator.

State

State is the module responsible for gathering information about the entire environment. This have earlier been described to be :

1. Power consumption
2. Active compute nodes
3. Number of virtual machines per compute node
4. Resources used
5. Importance

This information needs to be processed and presented in a way all three algorithms can use and understand.

Policy

The policy is the part and module of the prototype which will process the information presented by the state module. This will require an evaluation of the entire environment before it will be presented to the algorithm. When the algorithm starts processing the information it needs several arguments like number of buffer

or padding. Which is the spare hardware able to receive new virtual machines. The optimized results presented by the algorithm will be processed and stored.

Action

The preserved results from the algorithm will in this module be made into actions that will be taken in the environment. It can be as little as moving one virtual machine for a more optimized location or as much as moving hundreds of virtual machines and turning of 70% of the physical machines. The actions will typically be:

- Shutting down a physical machine
- Starting a physical machine
- Live migrating a virtual machine from one compute node to another
- Dividing a physical machine into not important and important CPU.

The illustration 4.2 is a more detailed overview of the different modules and information retrieved by the prototype.

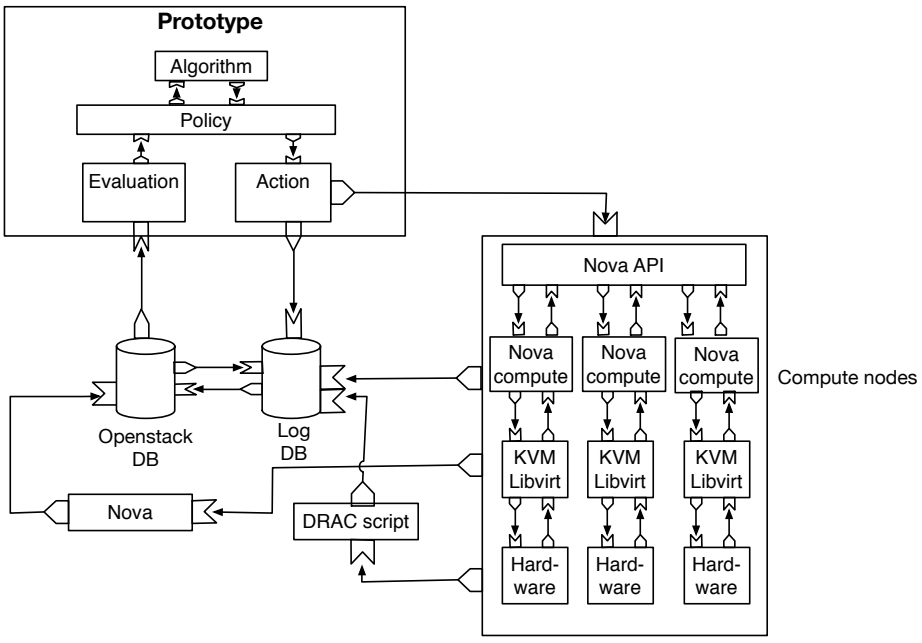

Figure 4.2: *BRIC model of the prototype to illustrate the process and controllers*

4.3 Applying bin packing to virtual machines

When applying bin packing to a cloud, the compute nodes (i.e the hypervisor servers containing the virtual machines) represent the bins in which to pack the virtual machines in such a way that we need as few bins as possible. The "space" of a bin, in which to pack virtual machines is defined by several constraints. The constraints can be arbitrary as long as they are reduced as virtual machines are placed in the bins. The number of constraints are also the dimensions of the bin. For example, the three constraints of a normal box are height, width and depth. For a compute node it may be CPU and memory or more.

Bin packing is a well known NP complete problem, this means that for real life adaptions, one has to choose algorithms that don't guarantee the most optimal solution, but close. In the design of a solution we are faced with two choices: the number and type of constraints and the algorithm for looking for the best solution.

Let C_n denote the nth compute node in a cloud environment. Virtual CPUs and memory are two obvious constraints of a hypervisor and are written as C_{n_VCPU} and C_{n_MEM} in capitals respectively. The current use of a constraints is written in non-capitals, like C_{n_vpcu}.

The ith virtual machine is represented as V_i. Every VM will consume an amount of the aforementioned constraints of a bin, such as VCPU and memory resources, denoted as V_{i_VCPU} and V_{i_MEM}. These two constraints are common for all three algorithms, however algorithm II and III add an additional constraint. Further definitions will be described in the result chapter for each algorithm.

4.4 Algorithm I: First fit bin packing

The first algorithm presented is also the most straight forward and presents the fundamentals of the process. Consider an array of N running compute nodes. The objective is to empty one and one compute node, giving it the name C_e. The compute node currently eligible for receiving the virtual machine is C_r. We start at the high end of the compute nodes, $C_e = C_N$, and try to find a place for each virtual machine starting at the other end $C_r = C_1$. As soon as the compute node with space is found, we place the VM there and pick the next VM on C_e and start at $C_r = C_1$ again. As soon as the compute node C_e is emptied it can be shut down. The algorithm then proceeds to $C_e = C_{N-1}$ and so on. If we find that we are trying to place a virtual machine into the same compute node, $C_e == C_r$, we have reached the end as there is no room below in the array.

All scripts developed in the process of getting a working prototype will be presented in the implementation chapter. They will be described in this chapter as pseudo code and for complete Perl scripts an interested reader is advised to see appendixes.

4.4.1 Formal notations

The formal notation requires a terminology to better understand the algorithm. The formal notations of the constraints will be described below. The figure below will illustrate a state of the system and as one can see the physical nodes will be called C, while virtual machines will be called VM_1 up to VM_i.

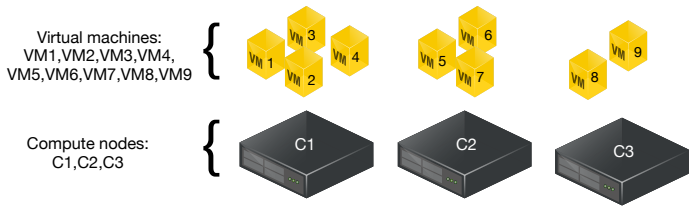

Figure 4.3: *Illustration showing the state of the system for formal notation.*

The figure 4.3 will be formally noted as:

C_1^t (VM_1^1, VM_2^1, VM_3^1, VM_4^1)
C_2^t (VM_5^2, VM_6^2, VM_7^2)
C_3^t (VM_8^3, VM_9^3)

C_3^t is the computer node number 3 and the t is the time. The formal notation of a compute nodes is C_n^t.

A VM in a compute node will be described as above VM_9^3 where the unique number of the VM is subscript in this case 9, and the location of the VM is superscript. The means that this VM is located at compute node number 3 and is VM number 9.

The number of running machines on a compute node will be formally notated as r_i, which makes the first VM notated as VM_1^n and the last VM as $VM_{r_j}^n$.

N is the maximum of physical nodes while n is the number of physical machines running. U is the utilization which means the number of physical running servers. This is defined by:

$$U = n \tag{4.1}$$

S is the number of machines not running. Calculated by:

$$S = N - n \tag{4.2}$$

The resources used by virtual machines is virtual CPU ($VCPU_i$) and memory used

by VM_i is $Vmem_i$. The total usage of memory:

$$Vmem_T = \sum_{x=1}^{r_j} Vmem_x \tag{4.3}$$

and the total usage of virtual CPU:

$$Vcpu_T = \sum_{x=1}^{r_j} Vcpu_x \tag{4.4}$$

The capacity of memory is written as C_nMem while the capacity of VCPU is C_nVCPU.

The terminology for utilization is:

$$U_nMem = \sum_{x=1}^{i} Vmem_x : Vmem_x \in C_n \tag{4.5}$$

Utilization of memory is the sum of the memory of all VM‘s if the VM is an element of the set C_n. This also applies to the VCPU. Written formaly as:

$$U_nVCPU = \sum_{x=1}^{i} VCPU_x : VCPU_x \in C_n \tag{4.6}$$

This will vary over time but utilization will always be smaller than capacity. The ground elements of bin packing is know in place, but this is exclusively the notations for the description of resource allocation at one certain time, which means that this will not make any decisions.

This is terminologies that will help us do calculations of how much resources that are used at all physical nodes, and how the VM‘s can be moved relative to the resources. A policy maximizing the effect of bin packing, with small or non consideration to performance but maximizing the effect by packing as hard as possible would save the system significant amount of energy but may compromise the performance. The threshold for memory T_{mem} will always be smaller than $\sum_{c=1}^{n} C_cmem$, and T_{VCPU} will be calculated by $\sum_{c=1}^{n} C_{VCPU}$.

Threshold memory:

$$T_{men} < \sum_{c=1}^{n} C_cmem \tag{4.7}$$

Threshold CPU:

$$T_{VCPU} < \sum_{c=1}^{n} C_cVCPU \tag{4.8}$$

Now as the essential is in place, these notations will be used in the following chapters to find a policy to optimize the environment.

4.4.2 The algorithm - policy

The policy requires an algorithms to know how to react to the state of the environment. The algorithm will be described be pseudocode.

This policy has one main function which is to move all running virtual machines to as few compute nodes as possible.

The policy should consider these constraints:

- Memory
- VCPU

The policy will need to measure all VMs and see if the compute node have sufficient with both memory and VCPU to receive a VM.

The policy will start at the highest number of compute node and start to move VM‘s from that node to the first node as long as there is enough VCPU and memory available at the first node. When the first node no longer have sufficient with VCPU or memory the system will start packing VMs to the second compute node. This can be compared with the Bin Packing technique "first fit".

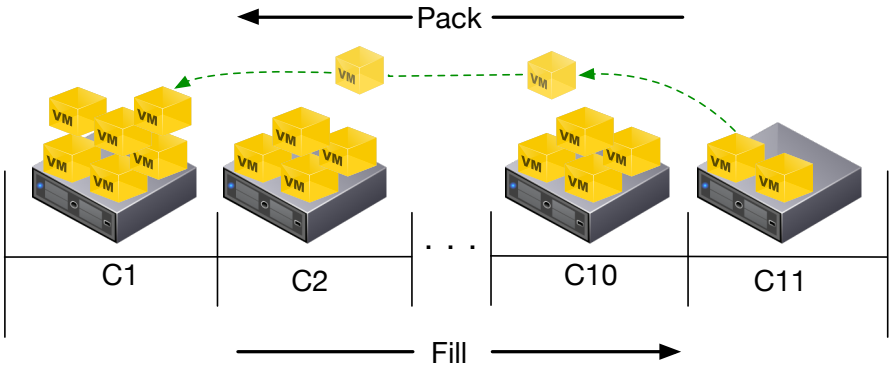

Figure 4.4: *The first fit algorithm starts from the end of the compute node array and looks for space from the start. From last to the first compute node.*

In pseudocode, the algorithm can be described like this:

```
FOR C_e in ( C_N, C_{N-1} ... C_1 ) DO
  FOREACH VM_e in C_e DO
    FOREACH C_r in (C_1, C_2, ... C_e) DO
      IF C_r == C_e THEN
        Finished()
      END
      IF (C_r_mem + V_e_mem ≤ C_r_MEM) and
         (C_r_vcpu + V_e_vcpu ≤ C_r_VCPU) THEN
```

```
            MigrateVM(VM_e,C_r)
            LAST
        END
      END
    END
    SHUTDOWN(C_e)
END
```

This pseudocode explains that for every compute node in the environment, starting with the compute node with the highest number, the prototype will try to move all virtual machines from that compute node to the node with the lowest number. This will be done if the compute node that is receiving the virtual machines have enough VCPU and memory the machine will be migrated to the target node. When the compute node sending a virtual machine is the same as the compute node that will receive the virtual machine the algorithm is finished. The empty compute nodes which are empty will be shutdown.

This will be the most basic policy which will demonstrate the features and prove that the policy will work. This policy lacks several features like booting machines if necessary, it will only maximize the bin packing without any other considerations. It will always start from the bottom and pack VM‘s to physical machine one.

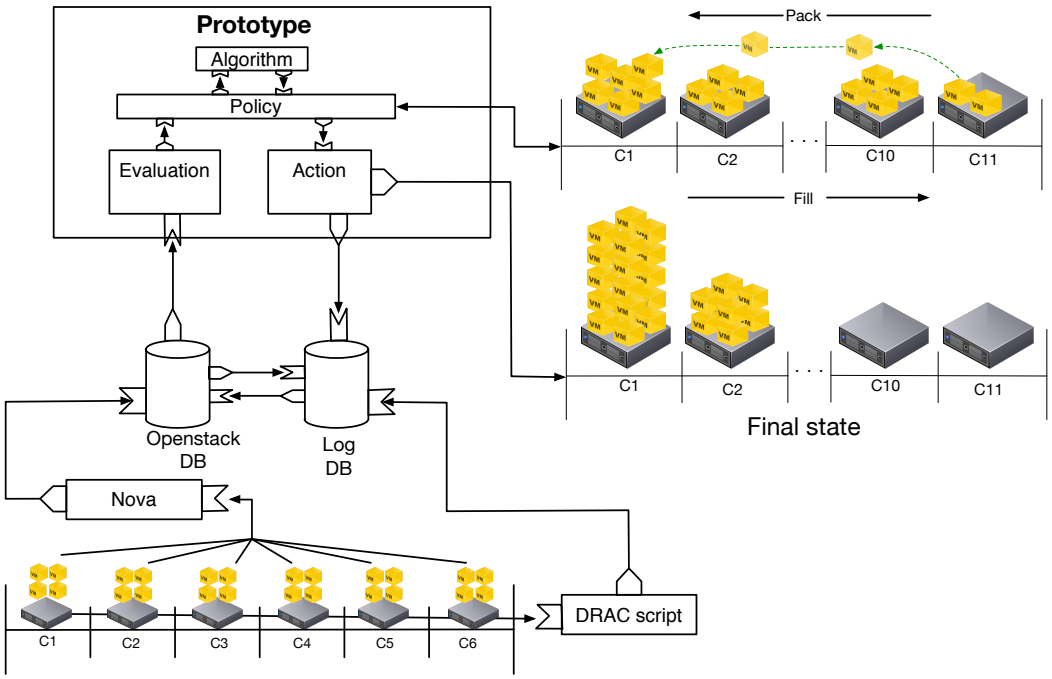

Figure 4.5: *BRIC model of policy 1, showing process and modules.*

4.4.3 Example

The section that follows is an example of how the calculations will be done and how all machines will be moved. In this example the N=3, which means 3 physical machines with seven virtual machines.

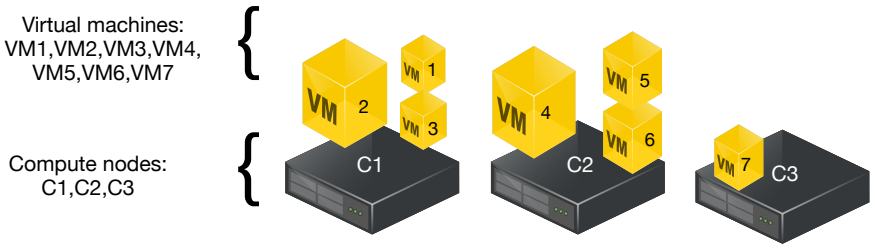

Figure 4.6: *Environment state at beginning of policy 1*

The virtual machines have these settings:

VM Name	Number of VCPU
VM_1^1	2 VCPU
VM_2^1	8 VCPU
VM_3^1	2 VCPU
VM_4^2	8 VCPU
VM_5^2	4 VCPU
VM_6^2	4 VCPU
VM_7^3	4 VCPU

OpenStack allows hardware to be overbooked to make room for more virtual machines. The overbooking of hardware can make constraints in regard to performance but since there is only a small chance that all machines are doing hardware demanding tasks OpenStack allows overbooking by 16. This means that the one physical machine with 64 CPU now can have $C_n VCPU$=16x64=1024 virtual CPU (VCPU). This is also done with memory although less than VCPU. Memory is overbooked by 1,6. In most environments there is no need to overbook since the capacity is sufficient to make one-to-one allocations. The example below will not overbook.
The prototype starts by collecting information from all VMs and from the compute nodes to find threshold values and the number of VCPU. To ease readability this example will not calculate memory.

$$C_{VCPU}^3 = 192 \tag{4.9}$$

Is the total capacity without overbooking. The next equation is the sum of the

VCPU for all virtual machines.

$$U^t_{VCPU} = \sum_{c=1}^{n} U_i = 32 \tag{4.10}$$

In this example there is a threshold of 30 percent. If the total amount of demanded hardware capacity for all virtual machine increases to over 70 percent, it will indicate that more physical servers need to be started. The algorithm will now start to move virtual machines optimizing the energy efficiency and see if it is possible to shutdown any compute nodes.

$$T_{VCPU} = 0,3 \tag{4.11}$$

At this time the virtual machines only use 50 percent of one physical machine, which means that all virtual machines can be moved into the first compute node. In this state, the environment is using only 16 % of the capacity. The prototype will now start at compute node C3 and start to see if there is possible to move VMs from C3 to C1. For each VM the policy will calculate this equation and if the it true the move will be prepared.

$$\sum_{x=1}^{i} VCPU_x : VCPU_t \in C_1 + VCPU_7 <= T_{VCPU^1} \tag{4.12}$$

If the first compute node runs out of capacity the algorithm will update the compute node indicator and the virtual machine that was to big for the first compute node will be placed at the next compute node. The next virtual machine will also be tried at the first compute node in case this one is smaller than the previous one and therefor can be place at compute01.

$$1 - \frac{U^t_{VCPU}}{C_{tn}} > T_{VCPU} \tag{4.13}$$

The prototype will always keep 30 percent threshold. If this threshold value is reached, it should boot a new machine. The finished state of the environment will be like illustrated in figure 4.7

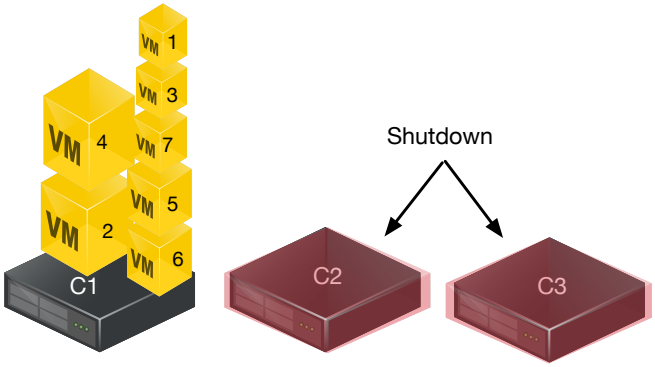

Figure 4.7: *This is an example of how a end state can be for policy 1.*

4.5 Algorithm II: 2D best fit bin packing with CPU zones and minimal migrations

The policy described in section 4.4 is illustrating both the proof of concept and the significant amount of energy possible to save by packing virtual machines with higher density, without increasing the amount of power consumed by the physical machine now hosting virtual machine in higher density than before. The algorithm only takes memory and CPU as constraints which means that the there are several other features like the number of live migrations and the importance of the virtual machines not taken into consideration. This section will explain how a more advanced policy with two dimensional bin packing can improve the policy by taking into consideration both how to decrease the number of live migrations and the importance of machines that always requires higher quality of service than other not so important virtual machines. The two additional features strongly increases the complexity by requiring both a way of decreasing the number of live migration and the need to divide physical hardware between important and not important virtual machines. This needed to be solved to ensure that a not important virtual machine not can compromise the performance of a important one.

Every compute node now has split its CPUs into two sets: low quality and high quality. As shown in figure 4.8.This is implemented using CPU affinity settings on the compute nodes for each virtual machine. A virtual machine that is classified as non-important will get the CPU affinity belonging to the low quality set while the important virtual machines are placed in the high quality set. As these two sets do not overlap, we basically get two separate constraints C_{n_LQVCPU} and C_{n_HQVCPU}, but a VM can only reduce the number in one of them. In effect, this becomes two 2D spaces with overlapping constraints.

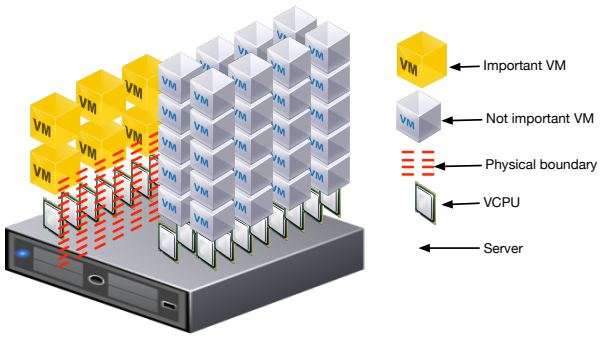

Figure 4.8: *Policy 2 overview of divided important and non-important virtual machines.*

4.5.1 Formal definitions

The formal notation will be the same as in policy P1, and will therefor only shortly be described in this chapter. The physical nodes will be called C, while virtual machines will be presented with a unique number and the location as VM_1^n for the first and the last VM as $VM_{r_j}^n$.

The resource usage of a virtual machine is $VCPU_i$ for CPU and memory as $Vmem_i$.
As policy 1 maximized the effect of bin packing as hard as possible, policy 2 will attempt to optimize the environment and algorithm to minimize the power consumption but also decrease the number of live migrations in compared to policy 1 and a higher level of quality of service.

To improve the number of live migration there are several important notations as the number of live migrations and the importance. The importance will be described as λ while all not important machines is α.

The constraints on the two CPU dimensions can be adjusted based on the desired amount of overbooking. For example, consider a compute node of 64 physical CPUs. Let the high quality set span from CPU 1 until 24, and the low quality set from 25 until 60, leaving the compute node itself with 4 dedicated CPUs. Since the low quality set should allow for a higher utilization of space, we can set the overbooking factor to 8:1, resulting in $C_{n\_LQVCPU} = 35 * 8 = 280$. For the high quality set we can use 2:1 and get: $C_{n\_LQVCPU} = 24 * 2 = 48$. How hard the non-important virtual machines will be packed have notation $D_{density}$, which means that D_4 will pack non-important machines 4 times harder than important one.

In order to reduce the number of migrations, we now traverse the array of compute nodes from the least populated to the most populated compute node.

4.5.2 The algorithm - policy

Similar to the policy 1 the new policy will first be described before the algorithm is presented as pseudokode, this code will be explained in detail before an example. This policy has the same main goal as the first algorithm but there is now added multiple constraints to the algorithm which will affect the way the virtual machines are packed into the compute nodes. These constraints will further increase the complexity of the work flow and the performance for each virtual machine. This policy has one main function just which is to move all running virtual machines to as few compute nodes as possible but now with additional constraints:

- Memory
- VCPU
- VMs levels of importance
- The number of live migration

Like the first policy this policy will moving virtual machines from right to left but this time the compute nodes will be sorted according to the number of running virtual machines. This is illustrated in figure 4.9 here the number of the compute nodes no longer is in ascending order.

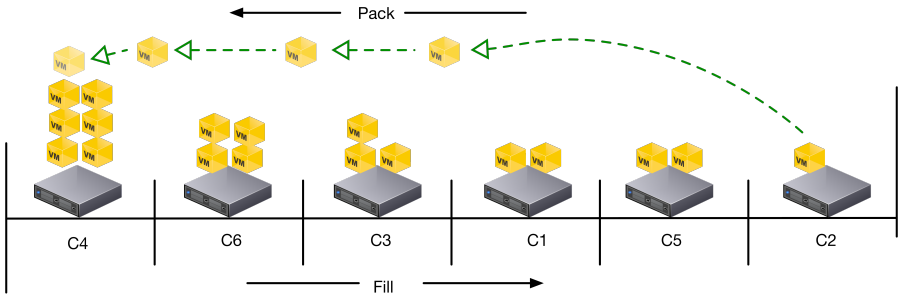

Figure 4.9: *Policy 2 with sorted compute nodes for live migration from right to left.*

Now as the compute nodes are receiving virtual machines the number of live migrations will be as low as possible. When one compute node receives a virtual machine it also sorts the machines according to the level of importance. Illustrated in figure 4.10.

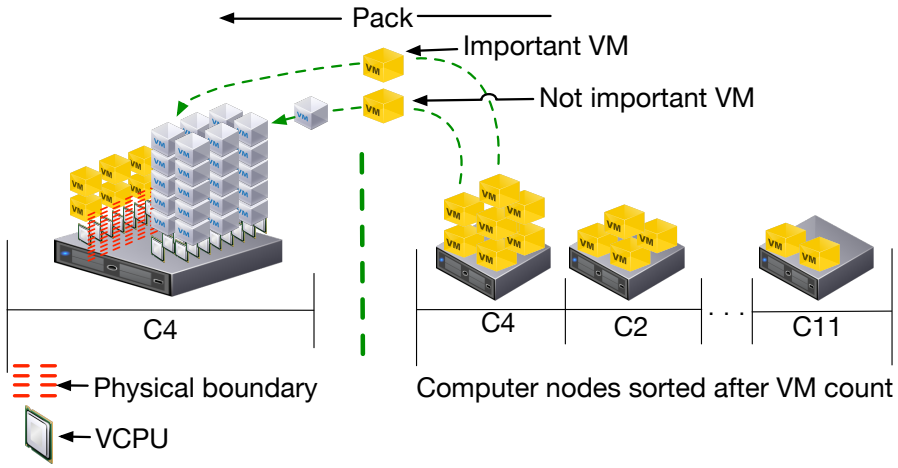

Figure 4.10: *Policy 2 in action, illustrating the live migration of virtual machines and the difference between important and not important virtual machines.*

After all possible live migrations have been done the algorithm stops and there should now be a more optimized environment for energy efficiency. The second policy will have more constraints than policy1 and this is illustrated in this BRIC modell figure 4.11.

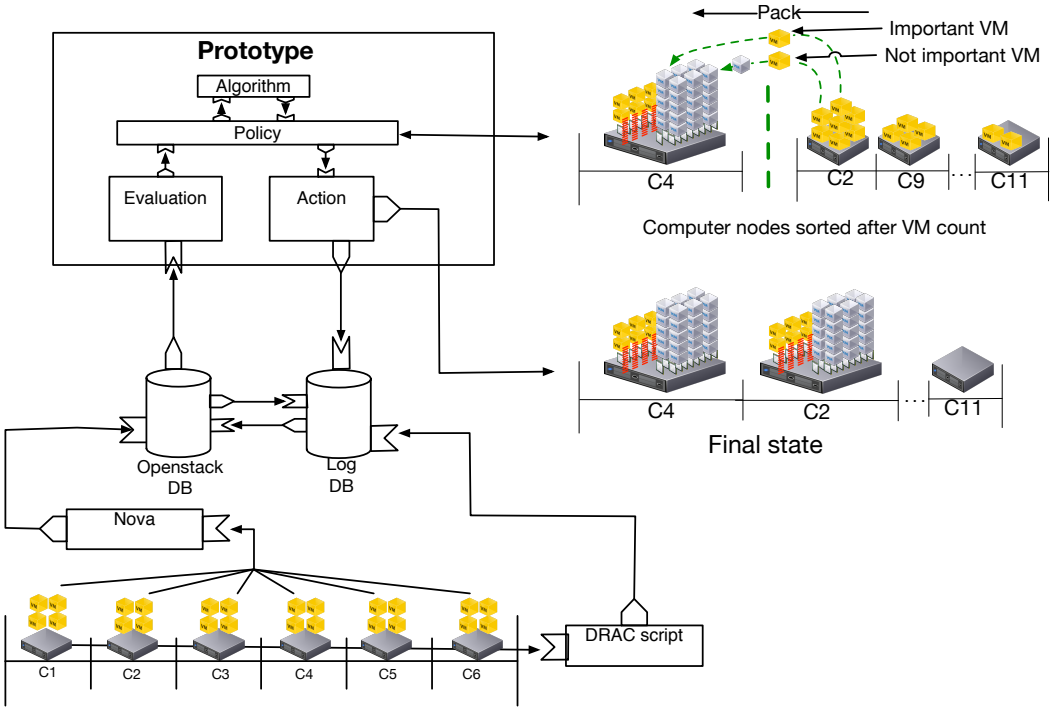

Figure 4.11: *BRIC model of policy 2 with the steps and actions done to the environment*

This is the pseudocode for the new algorithm:

```
FOR C_e in SORT('VM_COUNT','ASC',( C_N, C_{N-1} ... C_1 )) DO
  FOREACH VM_e in C_e DO
    FOREACH C_r in SORT('VM_COUNT','DESC',( C_N, C_{N-1} ... C_1 )) DO
      IF C_r == C_e THEN
        Finished()
      END
      IF isImportant(VM_e) THEN
        IF (C_r_mem + V_e_mem ≤ C_r_MEM) and
           (C_r_hqvcpu + V_e_vcpu ≤ C_r_HQVCPU) THEN
             MigrateVM(VM_e,C_r)
             LAST
        END
      ELSE
        IF (C_r_mem + V_e_mem ≤ C_r_MEM) and
           (C_r_lqvcpu + V_e_vcpu ≤ C_r_LQVCPU) THEN
             MigrateVM(VM_e,C_r)
             LAST
        END
      END
    END
```

```
  END
  SHUTDOWN(C_e)
END
```

To explain the pseudocode in words the first thing the algorithm will do is to sort all compute nodes according to the number of active virtual machines running. Then it will check if the compute node sending virtual machines is the same as the compute node receiving. If this is the case the algorithm is finished and the policy is done. If this is not true, the algorithm continues to check if the virtual machine evaluated for a live migrations is an important or non-important machine. This is done to see if there is enough room at the receiving compute node to host the virtual machine. The size of the machine will be determined by the algorithm depending on the importance level. If the receiving compute has room for the virtual machine it now migrates the virtual machine and updates the available space at the node sending and receiving. Then, if the compute node is empty is shuts down. The algorithm continues to the next compute node and starts over.

Policy 1 was a two dimensional algorithm with memory and VCPU as constraints, the new and improved is a three dimensional algorithm with additional constraints as importance.

4.5.3 Example

Policy 2 will in this section be done step by step in an example. The environment will have $N = 3$ as the environment for policy 1 but in this example there are more virtual machines.

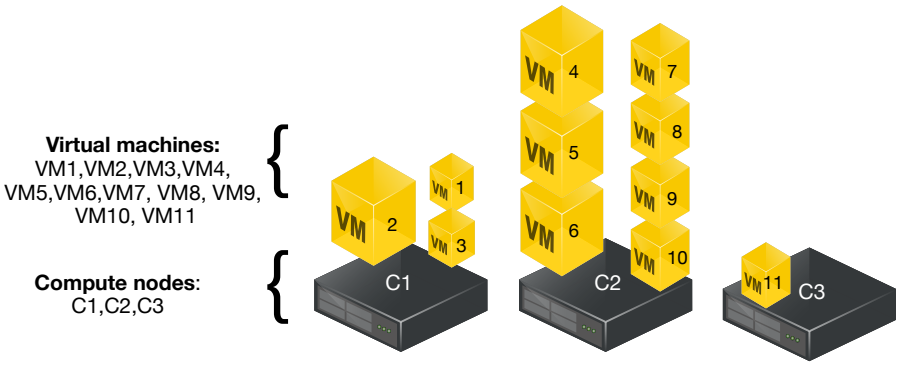

Figure 4.12: *Example environment for explaining policy 2.*

The virtual machine have these settings:

VM Name	Number of VCPU	Importance level
VM_1^1	1 VCPU	λ
VM_2^1	1 VCPU	λ
VM_3^1	16 VCPU	λ
VM_4^2	16 VCPU	λ
VM_5^2	16 VCPU	α
VM_6^2	16 VCPU	α
VM_7^2	4 VCPU	α
VM_8^2	4 VCPU	α
VM_9^2	2 VCPU	α
VM_{10}^2	2 VCPU	α
VM_{11}^3	1VCPU	λ

The policy starts by collecting information from all virtual machines and from the compute nodes. Then is finds threshold values and the number of VCPU. To ease readability this example will not calculate memory.

$$C_{VCPU}^3 = 192 \tag{4.14}$$

Is the total capacity without overbooking. The next equation is the sum of the VCPU for all virtual machines.

$$U_{VCPU}^t = \sum_{c=1}^{n} U_i = 79 \tag{4.15}$$

In this example the threshold is still of 30 percent. So if the total hardware demand is increases to over 70 it will boot new machines. The algorithm will now start to move virtual machines optimizing the energy efficiency and see if it is possible to shutdown any compute nodes.

$$T_{VCPU} = 0,3 \tag{4.16}$$

In this state, the virtual machines are using 43,75 % of the capacity. This is under the threshold and the policy will start to see if there is possible to optimize the environment. The prototype will now by sorting the compute nodes to minimize the number of live migrations. In this example C_2 have 7 virtual machines. If this policy 1 was conducted this would have generate 8 live migrations, since it would try to move all virtual machine to compute node 1. Policy however, will now start to move virtual machines to compute node 2 which only will generate 4 live migrations. This reduces the live migrations with 50%.

Figure 4.13 is an illustration of the importance.

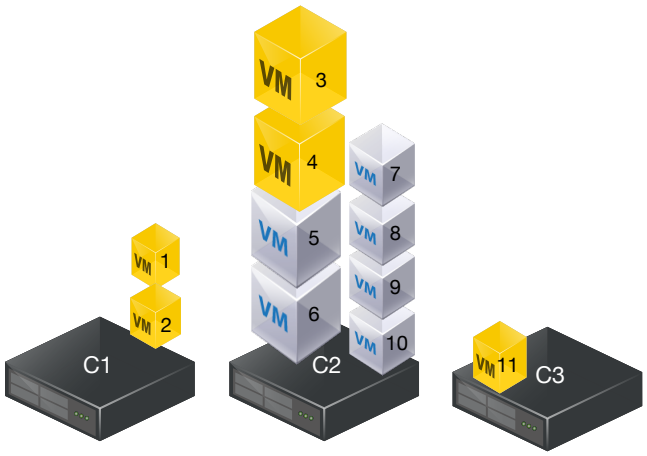

Figure 4.13: *This is an example of how virtual machines are divided into importance levels.*

$$C^1_{VCPU} = 64 \tag{4.17}$$

As one can see above the capacity of compute01 is 64 VCPU and the demand from the virtual machines in the environment is 84 all machines will not have enough space on one compute node without overbooking. With policy 1 this means that the original state would have to keep two compute nodes running, or reduce the performance for all virtual machine located at compute01. This is one of the main differences between policy 1 and policy 2. Since policy 2 has the ability to divide the compute nodes according to importance level all machines can be placed one compute node and only lower the performance of the non-important virtual machines. In this case the calculation of capacity would end up with:

$$ImportantVCPU = \lambda = 35 \tag{4.18}$$

$$Non - imortant = \alpha = (40/D_4) = 10 \tag{4.19}$$

By giving the prototype the opportunity to pack the virtual non-important virtual machine four times harder then the important one it is possible to still have room at only one compute node. The total capacity at one compute node is 64, but when the threshold is 30% it can only give 45 CPU kernels before the prototype will start two physical machines. In this example the environment now has a VCPU threshold of 30%.

$$1 - \frac{U^t_{VCPU}}{C_{tn}} > T_{VCPU} \tag{4.20}$$

The policy will be able to move all virtual machines to compute02 and still have 30% capacity left. This is illustrated in figure 4.14

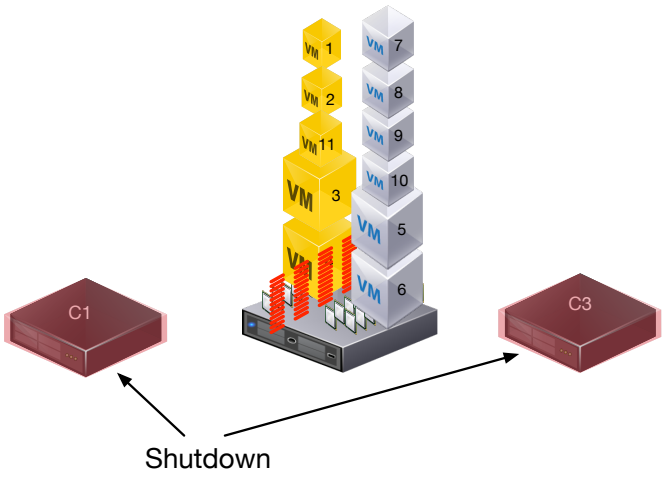

Figure 4.14: This is an example of how a end state can be for policy 2.

The purpose of policy 2 is as shown in figure 4.14 that it is possible to place more virtual machines at the same physical machine without performance constraints for important machines.

4.6 Algorithm III: 3D bin packing with importance as constraint

The last algorithm we present is based on seminal work of Martello et al. [21] for solving the three-dimensional bin packing algorithm. The algorithm resorts to a branch-and-bound optimization and was shown to exhibit fast convergence and to achieve near-optimal solution. To adapt the algorithm to our particular VM packing problem, VCPU and memory will be chosen to be the first two of the constraints. For the third constraint, defines a novel concept called importance capacity. Every virtual machine will have an importance weight associated with it.

When considering the dimensions for 3D bin packing they together create a much larger space in which to place virtual machines. Virtual machines can be placed next to each other in a 3D space, allowing them to "occupy" the same resource constraint multiple times. This is ultimately the challenge when transforming something that uses resources to something that uses space. [24] One therefore has to chose the dimensions with care. Using $VCPUxMEMxIMPORTANCE$ directly as dimensions would translate to $64x256x10$ which would fit a very high number of virtual machines.

In the experimental settings, a weight 2 is assign to an important virtual machine and 1 to a non important virtual machine. Intuitively speaking, the third constraint, importance capacity, permits to restrict the number of important machines co-existing in the same physical machines. The constraint is motivated by quality of service consideration in Cloud Computing. In fact, it is known that two VMs residing on the same physical machine and sharing the same physical CPU can interfere with each other resulting in a lower quality of service experienced by both virtual machines.

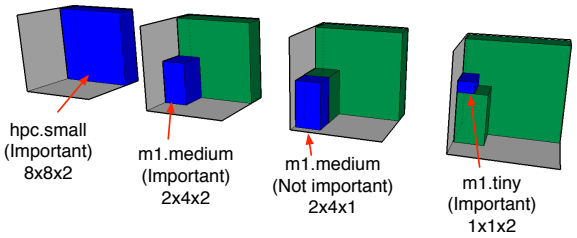

Figure 4.15: *This algorithm calculates a volume optimized solution regardless of previous location.*

For example, consider an important VM with 2 VCPUs, 4GB of memory. This would correspond to the VM being a cuboid of size $2x4x2$, taking up a volume of 16. In a bin of $8_{VCPU}x8_{MEM}x6_{IMPORTANCE}$ one would be able to fit 24 of them (or 48 non-important ones of the same type). For our compute nodes, $8x8x6$ was chosen as this best reflected the capacity we wanted.

Martello et al. [20] described the algorithm like this in pseudocode:

```
Copy input information to internal structures;
Compute lower bound L2;
Execute heuristics H1 and H2, and set u to the best solution value found;
WHILE no optional solution is found or stopping criterion is met DO
 Comment: perform the tree enumeration;
 assign the next box to an open bin or a new bin;
 check feasibility of the assignment with onebin_decision;
 IF the assignment is not feasible THEN backtrack;
 ELSE
  Check the current solution for (non-)optimality;
  check node limit and time limit;
  FOREACH open bin DO
  if the bin can be closed THEN
   compute a new lower bound and possibly backtrack
  END
 END
return best solution;
```

Two heuristic algorithms are used called H_1 and H_2, according to the descriptions given in Martello et al [20]. The first algorithm H_1 constructs number of layers of dimensions $W \times H \times d$,$(d \leq D)$ which in our case is VCPU, memory and importance. By solving a bin packing algorithm with one-dimension defined by the depth of the layers, the result is full bins of depth D. Algorithm H_2 fills every bin to maximize the volume filled. The result is a list containing a volume optimized solution for placement of virtual machines.

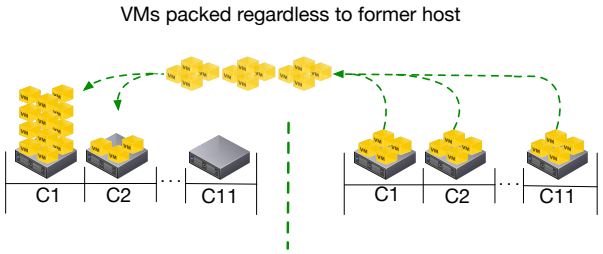

Figure 4.16: *This algorithm calculates a volume optimized solution regardless of previous location.*

One important assumption with this algorithm is that a non important virtual machine also will be an inactive virtual machine. This was not enforced in this algorithm, leaving the possibility that the performance levels of important virtual machines could be affected.

4.7 The environment

To get a working and realistic prototype the environment needs to be of a certain size. This project studies autonomic strategies to optimize the resources used in a data center and to address this challenge there is required a working cloud environment. More importantly the environment is required to have the ability to run live migration between the compute nodes. This technique has several requirements to the environment. Both the network and shared storage needs to be in place for a live migration. See OpenStack documentation for further description on setup.

The implementation of the prototype will be done on Norway's largest education cloud, which is built and hosted at Oslo university collage. The topology of the cloud is illustrated in figure 4.17.

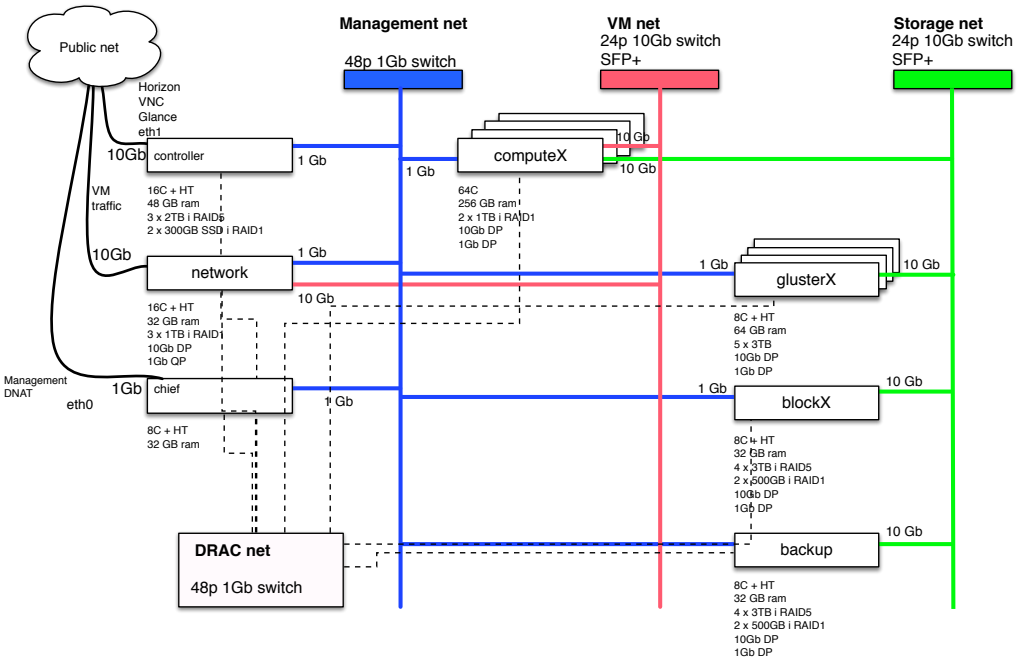

Figure 4.17: *Topology of the hardware architecture of the cloud created by the Network and System Administration department at Oslo and Akershus collage*

The entire cloud is placed in two full-scaled racks in a server room with proper ventilation and uninterrupted power supply. Rack number 2 has all compute nodes together with two switches for storage and VM net. Rack number 1 has all storage nodes, controller and chief machines along with DRAC switch used for gathering data for power consumption and management switch.
The compute nodes is dell poweredge R815 machines with 32 gigabyte memory and 64 cores processor. OpenStack allows these 64 CPU kernels to be overbooked by 16 which means it can be shared as 1024 VCPU for virtual machines.

As presented in figure 4.17 the architecture will consist of required elements from OpenStack which include shared storage and network to make it possible to live migrate one VM between physical nodes. These virtual machines will be controlled and managed by OpenStack which can be controlled from the web interface or form the command line. Examples will follow in the brief introduction to OpenStack chapter. When all physical requirements is in place and the live migration is operational will the implementation of the prototype begin.

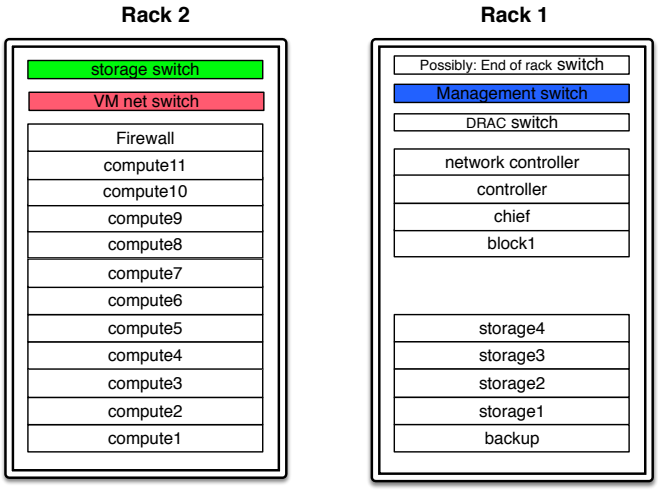

Figure 4.18: *The rack architecture of the cloud created by the Network and System Administration department at Oslo and Akershus collage*

Chapter 5

Result 2 - Implementation for experiments

This chapter will start by presenting three different algorithms. These will first be shortly described to give a clear overview of the main features the algorithm or policy possesses before a more detailed introduction to the formal definitions of the different aspects of the mathematical therms the algorithm solves. The algorithm will thereafter be described as pseudocode together with illustrations to give an simple graphical interpretation of complex mathematical algorithms. The details of the process will be presented both with text and a BRIC modell. To easily see the progress the algorithm is applied to a small example environment to show basic features. The section of the three algorithms will end with a summary of the experiments. After all three algorithms is described the implementation will be presented, before the results.

5.1 Implementation - Building the system

To measure the policies the need for monitoring of the system is present and the measure the power consumption is key feature for gathering of empirical evidence that the environment has a significant lower consumption of electricity after running the prototype than the original environment.

5.1.1 Monitoring power consumption

A perl script has been developed to connect to the DRAC card for consumption information for all machines.
The prototype is required to talk to mostly of the physical machines in the environment and it was seen as necessary to create public and private keys for the ability to send ssh commands between all machines, and to move freely between all machines without the password prompt for every connection.
The command for retrieving information about power consumption can be viewed on next page.

DRAC script

```

ssh root\@computenode racadm getconfig -g cfgServerPower

```

This command responds with 37 lines of information about the consumption. A section of the output can be view below.

DRAC response

```
cfgServerPowerLastMinAvg=285 AC W | 973 Btu/hr
cfgServerPowerLastHourAvg=281 AC W | 959 Btu/hr
cfgServerPowerLastDayAvg=280 AC W | 956 Btu/hr
cfgServerPowerLastWeekAvg=280 AC W | 956 Btu/hr
```

The information about power consumption generates 37 lines of information and the most interesting lines are:

- cfgServerActualPowerConsumption=276 AC W | 942 Btu/hr
- cfgServerMinPowerCapacity=422 AC W | 1439 Btu/hr
- cfgServerMaxPowerCapacity=827 AC W | 2824 Btu/hr
- cfgServerPowerLastMinAvg=285 AC W | 973 Btu/hr
- cfgServerPowerLastHourAvg=281 AC W | 959 Btu/hr
- cfgServerPowerLastDayAvg=280 AC W | 956 Btu/hr
- cfgServerPowerLastWeekAvg=280 AC W | 956 Btu/hr

These lines can tell us interesting information about every single physical machine to give us indications of when the machines have high peaks and average consumption both per min, hour, day and week. To make this as accurate as possible the script to get this information will be run every minute. This is to see if there is a difference in the actual power consumption when the virtual machines is for instance live migrated between compute nodes.

Since this information can give total control over every single kilo watt of electricity per hour used by the environment, the demand for a database is evident. This script will need to adjust every single line to give the OpenTSDB a input it can store in a meaningful way.

This is the script that edits every line for the database:

Line adjustment script for database

```
my @command = 'ssh root\@10.10.14.$i racadm getconfig -g cfgServerPower';
chomp(@command);
my $timestamp = getLoggingTime();
open (MYFILE, '>>data.txt');

foreach my $line (@command){

    $line =~ s/\s//g; # remove whitespace
```

```
9         $line =~ s/#//g; # remove # in sentence
10        $line =~ s/cfgServer//g; # remove cfgServer
11        $line =~ s/ACW//g; # remove ACW
12        $line =~ s/\|\d{1,7}(B|b)tu\/hr//g; # remove | 3-4 digit followed by Btu/hr
13
14        if($line =~ /Write-Only/){
15        }else{
16           print MYFILE "energy.server$line $timestamp host=compute$i\n";
```

This script runs through every single response line and makes changes like, removing white spaces, removing the hash sign and all other elements in the line not necessary for the database. The output will look like this for insertion to the database with timestamp and for which compute node the numbers are collected:

Power output for database

```
1  energy.serverPowerStatus=1 2014.03.19:18:36:30 host=compute1
2  energy.serverPowerAllocation=1400 2014.03.19:18:36:30 host=compute1
3  energy.serverActualPowerConsumption=274 2014.03.19:18:36:30 host=compute1
```

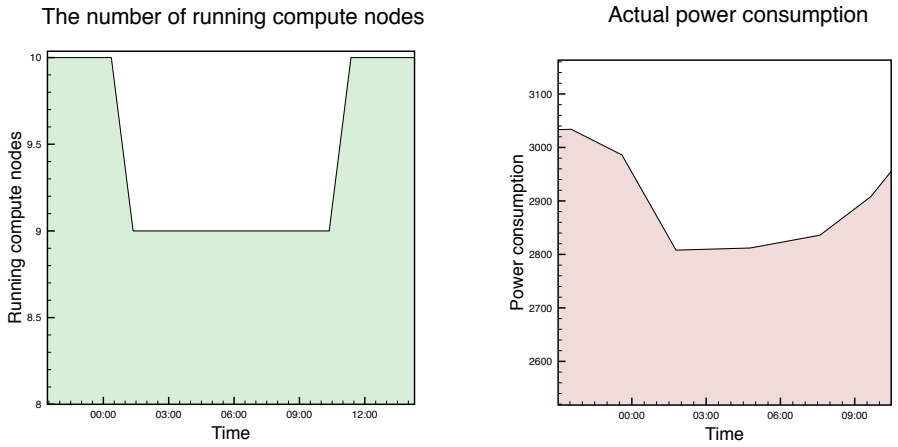

Figure 5.1: *Two graphs showing the number of compute nodes running and the power consumption*

The figure 5.1 illustrates the actual power consumption when the compute nodes goes from 10 to 9. When the compute nodes goes from 10 to 9 the power consumption goes from 3038 kWph to 2808 which is a 226 decrease.

The need for a database to handle all monitoring for later analysis is really applicable due to the number of lines generated and handled by the script. The monitoring of this environment generates 777 lines of information every time it runs. The script will run every minute throughout the day which will generate 1,1 million lines of power consumption information. The monitoring will continuously monitor the environment.

5.1.2 Monitoring of OpenStack - VCPU and Memory usage

Additionally to the monitoring of the consumption is it essential to monitor the environment in regard to:

- Free VCPU at compute node
- Free memory at compute node
- Total number of running virtual machines
- The number of running virtual machines at each compute node 6. The number of running compute nodes

This information is collected form the OpenStack database with these lines of code:

Retrieve information from OpenStack database

```
1
2   # Get the rows from database
3   my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username, $password)
4     || die "Could not connect to database: $DBI::errstr";
5   my $sth2 = $dbh2->prepare('select * from compute_nodes')
6     || die "$DBI::errstr";
7   $sth2->execute();
8
```

After collecting the information the results are processed line by line to find the information wanted. This is done by matching the lines after the names of the value:

Process information from OpenStack database

```
1
2   while (my $results2 = $sth2->fetchrow_hashref) {
3
4           my $compute_name = $results2->{hypervisor_hostname}; # get the hostname
5           my $compute_vcpu = $results2->{vcpus}; # VM number of cpu
6           my $compute_memory = $results2->{memory_mb}; # VM number of memory
7           my $compute_vcpu_used = $results2->{vcpus_used}; # VM number of vcpu used
8           my $compute_memory_used = $results2->{memory_mb_used}; # VM number of vcpu used
9           my $compute_running_vms = $results2->{running_vms}; # VM number of vcpu used
10          my $compute_free_memory = $compute_memory - $compute_memory_used;
11          my $compute_free_vcpu = 64 - $compute_vcpu_used;
12
13
14          $computestatus{$compute_name}{"running vms"}=$compute_running_vms;
15          $computestatus{$compute_name}{"compute_free_vcpu"}=$compute_free_vcpu;
16          $computestatus{$compute_name}{"compute_free_memory"}=$compute_free_memory;
17      }
18  }
```

All these values is inserted into a hashed hash in Perl and later modified by these lines to be adapted for the database. A hash is a data structure available in Perl for storing data. It can be associated with a map of your data, and all values in a hash can be retrieved and modified. The values in a hash is an un-ordered group of keys and values. These keys are unique strings that for instance can describe the values stored in the key. The value can be either a reference, number or letters. In this case hashes is used for storing data for the ability to later look up values by the keys.

Adapting of OpenStack information to OpenTSDB

```
1
2   my $prefix = "openstack.compute.";
3    foreach my $computenode (sort(keys %computestatus)) {
4
5       print "${prefix}compute_free_mem ".$
6       computestatus{$computenode}{"compute_free_memory"}."$timestamp host=$c$
7       print "${prefix}compute_free_vcpu ".$
8       computestatus{$computenode}{"compute_free_vcpu"}." $timestamp host=$co$
9       my $enabled = 0;
10      $enabled = 1 if ( $computestatus{$computenode}{"face"} eq ":-)" and
11      $computestatus{$computenode}{"enabled"}$
12      print "${prefix}enabled $enabled $timestamp host=$computenode\n";
```

The information from OpenStack generates three lines for insertion per compute node which means that it produces 33 lines for the database every time it runs, plus three lines with information of the environment as active virtual machines, migrations and total virtual machine count. The monitoring of OpenStack processes and inserts around 51 840 lines into the database every day. In total the monitoring and evidence collection is storing 117 216 lines of information of energy consumption and in total 169 056 every day and these lines is essential for the later analysis.

5.1.3 Building the masterscript

The prototype will be located at the physical machine "Chief" and from this machine can the software connect to all other machines and network cards in the environment. The software needs to collect information from:

- All compute nodes
- OpenStack database at Controller
- Nova controller at Controller for live migration commands

The main idea for this script is to collect all information from the environment and save this to the log for analysis before doing calculations based on the written algorithms. After the calculations done by the policy and algorithms the system will live migrate all machines according to the policy. Each live migration will be checked as successful before next machine can be migrated.

To achieve the main idea there is several fundamental elements that needed to be developed. The key for developing highly complex software is to break it down to minor elements that together can do significant improvements to optimize the environment. This is also how the prototype was built; multiple subroutines that later can be put together for greater good.

The first thing this script will do is to collect information from the OpenStack database to find the status of the environment by sending MYSQL database queries to the database located at Controller. Similar to the query for monitoring, this is done by these commands:

Retrieve information from OpenStack database

```
1
2   # Get the rows from database
3   my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username, $password)
4     || die "Could not connect to database: $DBI::errstr";
5   my $sth = $dbh->prepare('select * from instances where vm_state="active";')
6     || die "$DBI::errstr";
7   $sth2->execute();
8
```

The query sent to the database will select all information from the instance database table only for the virtual machines which has a state as "active". Active means that the machine is running, but it is not a indication that the virtual machine is doing anything productive.

The information received in the response from the database is then processed line by line for insertion into a hash. The purpose of a hash was described in the monitoring section, but in this case the information collected is in multiple levels which requires the use of a hashed hash function, where each value has multiple values that needs to be assigned to the first value. Each line retrieved by the database is processed to insert the values into the hash with unique keys for each value.

Process information from OpenStack database

```
1
2           my $vm_name = $results->{uuid}; # get the hostname
3       my $computenode = $results->{host}; # get the computenode
4       my $vm_vcpu = $results->{vcpus}; # VM number of cpu
5       my $vm_memory = $results->{memory_mb}; # VM number of memory
6
7
8        $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
9        $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
10       $status{$computenode}{$vm_name}{"project_id"}=$vm_project_id;
```

As seen in figure above the information required for later processes is hostname, location, number of VCPU for each vm, memory usage. These values is inserted into the hashed hash called status.

The same procedure is done for the compute nodes to retrieve information about each compute node. the query collects all information by asking to select all information from the table computenodes. The information is in the same way as for the instances inserted into a hashed hash which in this case is called computstatus. The information inserted is:

- Compute name
- The number of free VCPU
- The number of free memory
- The number of running virtual machines

The last section of information the prototype requires is the state of the compute node. To collect this information a connection to the Controller is established to gain information about compute nodes. This command collects information directly from OpenStack nova:

Process information from OpenStack Nova

```
1 ssh controller nova-manage service list
```

The command will respond with information which is used to categories the compute nodes into three different states. A compute node can either by running, stopped or having an error. This gives the script the ability to sort compute nodes to know which of the nodes that can be started or stopped if the environment fulfills the description for either shutting down or booting up physical machines.

Now as all required information is collected the processing can begin.

The first thing is to preserve the state of the environment due to the check that is done after the policy has executed to see if there is any changes done to the environment. The preserving is done by saving the hashed hash storing the values for the location of each virtual machine to a new hashed hash called the vm_{old_state}.

5.2 Policy 1 - Algorithm I: Simple first fit bin packing

All prerequisites for the policy is now fulfilled and the policy can start the calculations of the environment. This policy is as already described a algorithm that aggressively relocate machines from the compute node with the highest number to the lowest. In principle it starts form the compute node with highest number and for each single virtual machine the policy checks if the first compute node has enough VCPU and memory to host the virtual machine and if it does it updates the location in the hash for later live migration. Then the policy goes to the next virtual machine to do the same check. This loop will not stop until the compute node the policy tries to empty is the same as the receiver. The policy is aware of that virtual machines can have different sizes, and is therefor obligated to test the first compute node for every single virtual machine. This is to make sure that the first compute node is completely filled even tough one machine did not fit.

This is possible by making for and foreach loops that runs through all virtual machines starting at the compute node marked as vacanter for the compute node with the highest number which is not empty.

Vacanter vs target

```
1 for ( my $i = $VACANTER; $i >= $TARGET; $i-- ){
```

This for loop will be true as long as i which is the vacanter is larger or equal to the target. The vacanter will be decreased by one when the loop is finished. Foreach virtual machine inside this for loop which is the highest compute node it will take all virtual machines and try to replace it to the target node.

For each vm in the compute node

```
1  foreach my $vm ( keys %{$status{$computenode}}){
```

This foreach loop will collect information from the earlier assembled hash with all virtual machines from the specified compute node which is called the vacanter. To keep track for which virtual machine the prototype is processing this will be written to screen for information.

Thereafter the prototype requires another for loop to ensure that the script always for every virtual machine will start at the first target node. This will be as described above, a for loop inside a foreach loop for every virtual machines for all compute nodes.

For loop for j equals target

```
1  for ( my $j = $TARGET; $j <= $i; $j++ ){
```

The for loop will ensure that the $\$j$ or target will be less than or equal to $\$i$ which is the vacanter before increasing the target node.

The prototype is now ready to do the actual testing which will decide if the virtual machine located at the vacanter can be relocated to the target node. This is tested by collecting and calculating if the compute nodes free VCPU is larger or equal to the VCPU required of the virtual machine.

CPU test

```
1  if ( $computestatus{$targetnode}{"compute_free_vcpu"} >= ($status{$computenode}{$vm}{"vcpu"}) $
2          $cpu_ok = 1;
3      }
```

This is where the prototype will decide if the VCPU should be one to one or if the VCPU can be overbooked. As discussed earlier in the project OpenStack approves overbooking up to 16 times. This means in practice that 16 virtual machines shares one VCPU. For this experiment the VCPu is set to be one-to-one.

The same procedure will now be calculated for the memory.

If both VCPU and memory did fit into the target node the location to the virtual machine would be updated for later live migration. If the virtual machine did not fit into the target node the for loop would update the target node by going to the next compute node. Then the prototype will do the exact same procedure as described above until the virtual machine has been relocated or target node is the same as the vacanter. The last loop will not stop until all target nodes are filled and all virtual machines at vacanters has been moved. The hash will now have an updated list for all virtual machines with the compute node it has been relocated.

Relocation - action module

The policy have done the calculation of where the virtual machine should be located to optimize the environment according to the power consumption. The last part of the policy is a important part which handles the actual moving of the virtual machines while they are still running.

The routine that handles the moving will connect to the controller and ask for a live migration for one virtual machine:

The actual live migration command

```
1 ssh -t controller "source creds; nova live-migration ad1e2027-xxxx-xxxx-xxxx-872fc086e7dd compute06
```

The command requires information as the uuid of the virtual machine and the target compute node which the virtual machine will be move to. This information is stored at the hashed hash values under the key name uuid. The software needs to see if there is a difference between the old compute node and the new location. If there is a difference the virtual machine can be relocated.

Run migrations

```
1  foreach my $vm ( keys %vms){
2
3      if ( $vms{$vm} ne $vms_old{$vm} ){
4         print "Migrating $nametable{$vm}: $vms_old{$vm} -> $vms{$vm}\n";
5               system("ssh -t controller 'source creds; nova live-migration $vm $vms{$vm}'");
6            while( getVMLocation($vm) ne $vms{$vm}){
7               verbose("sleeping...\n");
8               sleep $MIGRATIONPAUSE;
9            }
10        }
```

The software will not move to the next virtual machine until the live migration is confirmed as complete.

Scaling of physical servers

After all migrations has been completed the system is now moved into an new state. This state has packed as much as possible to the compute nodes with the lowest number. The last part of the script will be to see if there is possible to shutdown or scale up the numbers of running physical servers. I feature necessary for the prototype is the ability to have a buffer which will ensure that the environment always can handle a sudden change in the number of virtual machines. In this policy the buffer of running but empty compute nodes can be given as an argument to the policy. This will create a point where the script stop the algorithm that moves the virtual machines which is the crosspoint for vacanter and target plus the buffer given as an argument.

Running scale down

```
1
2     if( ($crosspoint + $buffer) < $maxrunningcompute){
3        print "We are using too many compute nodes, scaling down\n";
4        for ( my $i = ( $crosspoint + $buffer + 1 ); $i <= $maxrunningcompute; $i++ ){
5           my $number = $i;
6           $number = "0" . $number if $number < 10;
```

```
7           my $computename = "compute$number";
8           print "shutting down $computename\n";
9           system("ssh $computename service nova-compute stop");
10      }
```

The same test is executed to see if there is to few running machines:

Running scale up

```
1
2       }elsif(($crosspoint + $buffer) > $maxrunningcompute){
3         print "we are under capacity, scaling up\n";
4         my $number = $maxrunningcompute + 1;
5         $number = "0" . $number if $number < 10;
6         my $computename = "compute$number";
7         print "starting $computename\n";
8         system("ssh $computename service nova-compute start");
9
10      }
```

Figure 5.5 is an illustration of the prototype and the state of the system from start to finish.

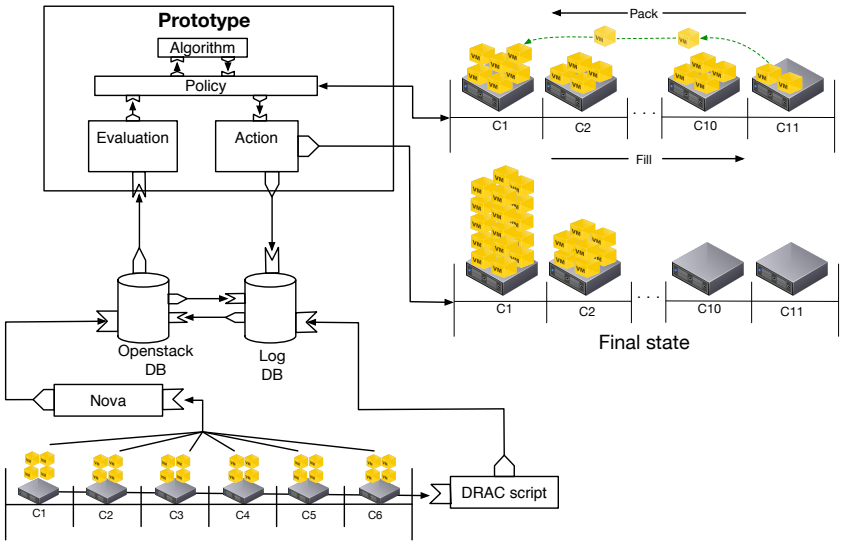

Figure 5.2: *policy P1 implementation*

5.3 Policy 2 - Algorithm II: 2D best fit bin packing with CPU zones and minimal migrations

The environment is the exact same as for policy P1, which means that the monitoring and data collection was done as described in section 5.5. Both the physical conditions and software conditions where monitored and all data inserted into OpenTSDB for later analysis.

This prototype will as policy P1 be located at the "Chief" machine in the environment. The developement of policy P1 was done in regards to the ease of reusing the basic modules of the script with different algorithms. The benefit of building a script with multiple sub routines can now be obtained be reusing the different modules already built. These modules reused in P2 are the collection of information regarding the environment, and the insertion of the values into hashed hash with unique keys to easily can be retrieved later in the prototype. For more details an interested reader is advised to return to chapter 5.5.3 or see complete scripts in appendix.

By using already built software the prerequisites for the policy is now completed by having complete state of the environment, and the policy can now start the work flow for a more optimized environment. P1 started by moving the virtual machines located at the compute node with the highest number to the compute node with the lowest number, but with this policy the need for restricting the number of live migration is vital. To ensure that the lowest number of live migration is obtained the policy starts by sorting the compute nodes in descending order. Values stored in a hash in in no way sorted and there is no way of sorting the hash storage and preserve the order. This means that the sorting needs to be done when needed and stored in a array. By storing the order in a array it is possible to use the first fit three dimensional algorithms approach to achieve the lowest number of migrations.

As seen in figure 6.1 the sorting of the compute nodes is in descending order to ensure that the number of migrations is as low as possible before starting to move virtual machines. The sorting of compute nodes requires a foreach loop to look through all compute nodes to sort and print the order. This can be done by sorting on the hash key vmcount as seen below.

Sort the compute nodes

```
1  foreach my $computenode (sort { $vm_count{$b} <=> $vm_count{$a} } keys %vm_count) {
```

To ensure the correct order the is collected from the array storing the compute name. The vacanter and target defined before entering the next foreach loop.

Storing the sorted nodes

```
1      $targetnode = @running_compute[$t];
2      $vacanter = @running_compute[$v];
3      $last_compute = $computenode;
```

To ensure that the loop has a end all compute nodes that receives virtual machines is stored into a new hash. This is later check by the prototype that the algorithm is finished if the vacanter have received virtual machines. This indicates that the target and vacanter is the same compute node.

A significant change to this policy is the levels of importance. Before any changes and policy actions is taken the policy required a overview of the environment in regard to how many virtual machines that are important and not important. This information is vital for the calculation of running compute nodes and the replacement of virtual machines.

Like P1 the policy now tries to move all virtual machines located at the vacanter to the target, and this is done by checking if the target node have enough space for the virtual machine.

Not important virtual machines will in this policy be packed with a higher density than important virtual machines. The important virtual machines will be given a one-to-one VCPU to ensure that they always have the performance ask for, while not important virtual machines will be packed with a density equal to eight-to-one VCPU. This means that the virtual machine believes that it has one virtual to itself but in fact the virtual machine shares the VCPU with eight other virtual machines also believing they have it alone.

This was in practice solved by adding the names of not important virtual machines in a file. The reason for adding not important is to ensure that no virtual machine do not get performance weaknesses without asked for it.

If a virtual machine is important or not is verified when the prototype checks if the compute node have enough space to host the virtual machine. As described above the main difference is the VCPU allocation.

Checking for importance

```
1  if ($vm ~~ @not_important) {
2              $vcpu_nr = ($status{$vacanter}{$vm}{"vcpu"});
3              $vcpu_subtract = ($vcpu_nr / 8);
4              # print "Not important VM detected: " . $vm . "VCPU: " . ($vcpu_nr+1) . " \n";
5          }else{
6              $vcpu_nr = ($status{$vacanter}{$vm}{"vcpu"});
7              $vcpu_subtract = ($vcpu_nr);
8              #  print "IMPORTANT:  " . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
```

This code decides if the virtual machines should allocate one-to-one VCPU or one-to-eight. The physical restrictions was designed to be enforced by virsh and KVM.

As illustrated in 5.8 KVM assigns physical hardware to each virtual machine by deciding which CPU the virtual machine can talk to. By letting not important virtual machine only talk to not important allocated CPU, the prototype ensures that important virtual machines has physical CPU allocation according to the configuration. The important virtual machines is in the same way as not important virtual machine receiving commands instructing the virtual machine to which CPU they can talk to. This can be compared to have different enclosures to different types of animals.

The commands controlling the sizes and cross points to the enclosure can be viewed below.

Determine size of VM

```
1  if ($vm ~~ @not_important) {
2      my $i = 0;
3      while ($i <= $vcpu_nr){
4              # system("ssh $node virsh vcpupin $line $i $low_cpu");
5             # print "\tssh $targetnode virsh vcpupin $vm $i $low_cpu \n";
6            $commands{$targetnode} .= "virsh vcpupin $vm $i low_cpu; ";
```

```
7       $i++;
8   }
```

This code saves all virsh commands to a hash value for sendig the commands after all virtual machines have been love migrated. An important detail is the fact that virsh requires a individual command for each VCPU dedicated to the virtual machine. This means that if the virtual machines has eight VPCU, the prototype is required to send eight virsh commands to the compute node. Illustration 6.3 shows the process of live migration of virtual machines according to levels of importance. To ease the process of CPU affinity the virsh commands was separated form the main script. This was done to give the long term test more abilities than required for the single experiment. The entire virsh script is included as an appendix for interested readers. This script can take these arguments.

- -c to print CPU affinity for all virtual machines at all compute nodes running
- -a to divide all virtual machines in HQ and LQ, must give -L low quality as a range as $4 - 25$ and $-H$ high quality $26 - 63$
- -u for uuid to CPU pin, must give $-r$ range for specific uuid

The policy has now done the scheduling of the virtual machines and updated the location to the hash values for later live migrations. The policy now reuses the sub routines from P1 and start to live migrate the virtual machines. For each migration the prototype checks for successful migration before moving to the next virtual machine. The scaling of the physical machines happens after the relocation.

5.4 Policy 3 - Algorithm III: 3D bin packing with importance as constraint

The last algorithm implemented is based on semial work of Martello et al. [21] for solving this challenge as a three-dimensional bin packing problem. The algorithm will optimize the same environment as policy one and two. As earlier the both the physical conditions and software conditions where monitored and all data are inserted into OpenTSDB for later analysis.

This prototype will be located at "Chief" machine, and the script used already built modules from policy one and two. This algorithm as earlier described resorts to a branch-and-bound optimization and was shown to achieve near optimal solution. To adopt this algorithm to our particular VM packing problem VCPU, memory is the two first constraints while the last constraint is the importance. This is what this algorithm requires to work properly. All virtual machines in the environment needs to be categorized and inserted into a file for the algorithm to read and process. The algorithm reads and calculates new locations for all virtual machines to optimize the space required. These new locations are stored for later live migrations. The

implementation of this algorithm will now be explained.

This algorithm stands out compared to the two other algorithm already presented and implemented by not using non of the information about locations of the virtual machines. Algorithm 3 only cares about the size of the virtual machine and not the previous location. This information is required to be according to the file format the algorithm can understand.

File format for algorithm 3

```
203 8 8 6
1 1 2
2 4 2
2 4 2
2 4 2
1 1 2
2 1 2
2 1 2
```

The first line in the file, first lets the algorithm know how many items or virtual machines there is in the environment. Followed by the size of the bins or in this case the available space at a compute node. This space is divided into height, width and depth. All lines following is the sizes of the virtual machines. Line two is for the first virtual machine and the next is for virtual machine number two. This goes on until all virtual machines are added into the file.

The information gathered from the environment is collected from the database as earlier described as the get status sub routine. For each virtual machine the importance level i check by simply reading through the file containing the UUID for all not important virtual machines.

Check for importance level

```

foreach my $computenode (keys %status) {
    foreach my $vm (keys %{$status{$computenode}}) {
        if ($vm ~~ @not_important) {
            $implevel = "1";
            }else{
            $implevel = "2";

        }
```

The levels of importance can easily be changed by replacing implevel with the wanted level of importance. Since the bins or physical machines have a lower amount of space the VCPU stands as it is, but the memory needs to be changed from thousands to single digit. It could be possible to hardcode the difference but the it is better to make sure that the memory levels can be matched. This was done by simply matching the amount of memory and change the amount.

Changing the memory size

```
        $mem = $status{$computenode}{$vm}{'memory'};
        if ($mem =~ /512/){
            $mem = 1;
        }elsif($mem =~ /2048/){
            $mem=2;
        }elsif($mem =~ /4096/){
```

```
7           $mem=4;
8       }elsif($mem =~ /8192/){
9           $mem=8;
10      }elsif($mem =~ /16384/){
11          $mem=16;
12      }
13
```

All information is now retrieved from the hash storing all virtual machines and is inserted into the file like this:

Matching for the memory

```
1       print FILE $status{$computenode}{$vm}{'vcpu'} . " " . $mem . " " . $implevel . "\n";
```

A drawback with this file is that there is not possible to know which object in the file that is belonging to which virtual machine. This can create a situation where there is no way to know where to move the virtual machines since the name of the machine cant be stored in the file. This challenge was solved by inserting the virtual machine name in a hash together with the place that machine got in the file. Like this:

Storing the names and place in file

```
1   $order{$vm}{"plass"}=$k;
2       $k++;
```

The file now contains all information required from the environment. The algorithm which is a callable C-code now requires to be compiled by following command:

Compiling the C code

```
1
2   gcc -ansi -o 3dbpp -O5 3dbpp.c test3dbpp.c -lm
3
```

By compiling with the test3dbpp.c configuration the c program can run with the file as an argument. The program is piped together with several commands to get a readable response that can be used to find the new locations.

Executing the algorithm

```
1   `./3dbpp file4.txt 0 0 0 0 | grep Bin | cut -f 2 -d ":" | cut -f 4 -d " "`;
```

The response from the program is captured by the software and processed to find new optimized locations for the virtual machines.

Processing the response

```
1   foreach my $vm (keys %order) {
2       foreach my $nr (keys %{$order{$vm}}) {
3           # print "\t$vm: " . "plass=". $order{$vm}{'plass'} . "\n";
4           my $plass = $order{$vm}{'plass'};
5           my $res =  @response[$plass];
6           my $nodecode = "compute0$res";
7           chomp($nodecode);
8
9           if( getVMLocation($vm) ne $nodecode){
10              print  "$vm:  from $old_spot to $nodecode\n";
```

```
11              $live++;
12          }
13      }
14  }
```

When the output is placed into an array called response the software takes each new location and compares this location to the previous one according to placement in the file. It runs through all virtual machines in the hash order and computes the new location and checks if this is the same as before. If the location has been changed this is counted as a live migration.

A clear drawback with this algorithm is that it can generate new locations for all virtual machines for every run.

Chapter 6

Result 3: Measurements and analysis

This chapter covers results and analysis from all experiments. Experiments regarding the environment is first presented before all algorithms is presented in ascending order. Results from experiments regarding algorithms will first be presented in form of graphs to ease the readability followed by further analysis of key values. The algorithms will first be tested in the environment illustrated in graphs before different scenarios are simulated.

6.1 Testing the environment and prerequisites

Before any proof of concepts could be done there where several key features required to work. This is a list of the experiments performed to ensure this prototype could be built. Task - Testing the environment:

- Task T_1 Live migrate a virtual machine
- Task T_2 Check connection to databases and information regarding state of environment
- Task T_3 Check monitoring and data collection
- Task T_4 Test sending shutdown command to a compute node
- Task T_5 Test starting a compute node from IDRAC 6 interface
- Task T_6 Calculate the time a start up for a compute node

The live migration of virtual machine T_1 is the heart of the prototype and is a key feature for the prototype. Live migration worked as intended to all compute node and was tested by moving one machine through all compute nodes. Further testing was the time each machine needed to get from one compute node to the other. T_1 proved that the time each machine required depended on the numbers of VCPU and memory. A feature added to the scripts was a timer which checked the database every third second to see if the migration was successful. The mostly used virtual machine type contains two VPCU and four GB of memory. These machine took in

average 9 seconds to migrate from start to end including security buffer of a couple seconds to ensure not provoking failures by stress in the system. Migration of the next machine started as soon as the first one was completed.

The information regarding the environment and state of all compute nodes is collected from the database T_2 and it was therefor a key feature to have a programming language capable and reliable with regarding connections to the database. Task T_2 showed a 100 % success rate with the Perl module DBI. Information was retrieved from the database in a orderly matter for easily be changed in the script.

The monitoring part T_3 of the environment is essential for the autonomic feature of the prototype. Collecting of the information inserted in the openTSD database required to be in orderly matter due to the machine learning part of the system in order to scale physical machines either up or down. OpenTSD also includes graphical interface so an system administrator easily keep track of the system, without any manually interference .

All commands performing actions to the environment was individually tested to ensure that every single line of code completed the task without any additional features or problems. Every module in the prototype was tested and worked according to plan after some adjustments. Some of the important features like start T_4 and stop T_5 command to the compute nodes. These experiments went according to the plan and the shutdown of an empty compute node happen in seconds.

T_5 which started a physical machine was performed by sending a command to the IDRAC interface which is the exact same as pushing the power button of the physical machine. The experiment showed that the average start time of a compute node is:

Startup time:	3.08 minutes

6.2 The experiments of policy 1

After all prerequisites and modules was tested in the environment, the experiments regarding the proof of concept and algorithm 1 could be started. The main goal of the first algorithm was to give proof of concept that it is possible to reduce the consumed energy in the environment by packing virtual machines harder to decrease the amount of idle hardware as in compute nodes.

This is the experiments conducted of algorithm 1:

- E_1 Simulate test run of algorithm and calculations
- E_2 Run policy 1 with 1:1

The first simulation E_1 was conducted to see how the environment would react to the changes made by the algorithms and to see how the calculations done by the

script could influence the state of the system. The experiment showed a drastic decrease of running compute nodes by moving all virtual machines to the lowest compute node with enough space for the virtual machine. The actual migrations of the machines was not conducted but the simulation gave a good insight into the algorithm and the calculations. After some adjustments to the calculation in regards to the number of active virtual machines and not virtual machines stopped and deleted, the algorithm was ready for the first proof of concept.

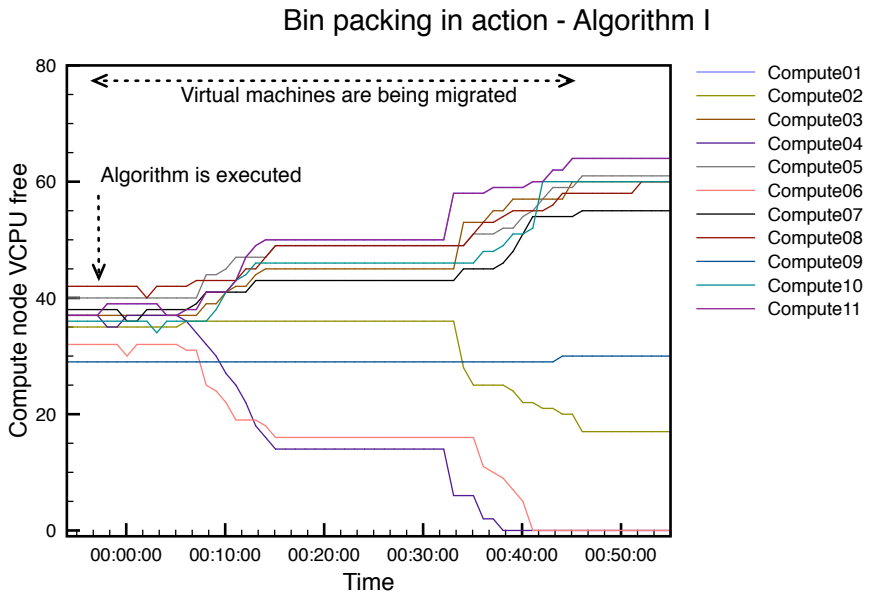

Figure 6.1: *Data showing the first run of algorithm 1 executed on the cloud environment with no prior packing. 1:1 VCPU*

Figure 6.1 represents experiment E_2 and illustrates compute nodes and the number of free VCPU at each compute node. When the experiment was conducted there were 225 virtual machines running in the environment. The figure shows a balanced environment where all compute nodes have between 35 to 45 free VCPU before the algorithm is conducted. The algorithm starts by calculating new locations for the virtual machines which are all treated equally, and will be moved to optimize the environment according to power consumption. After the calculations there is possible to see in the graph that the virtual machines are starting to move from compute node to compute nodes with the lowest number. The graph shows the number of free VCPU at each compute nodes which means that the physical machines receiving virtual machines will drop in the graph since the number of free VCPU is decreasing.

Fifteen minutes into the test the policy, an interesting event happend. A virtual machine live migrated from one compute node was not received by the target compute node which caused the policy to stop. The virtual machine was sent from the compute node and received by the target node but Openstack did not notice the migration which led to a freeze, but by manually update this in Openstack the policy

could continue.

The result was an environment that went from 11 compute nodes to four. As one can see in figure 6.1 the end state was a environment running compute nodes compute01 to compute04.

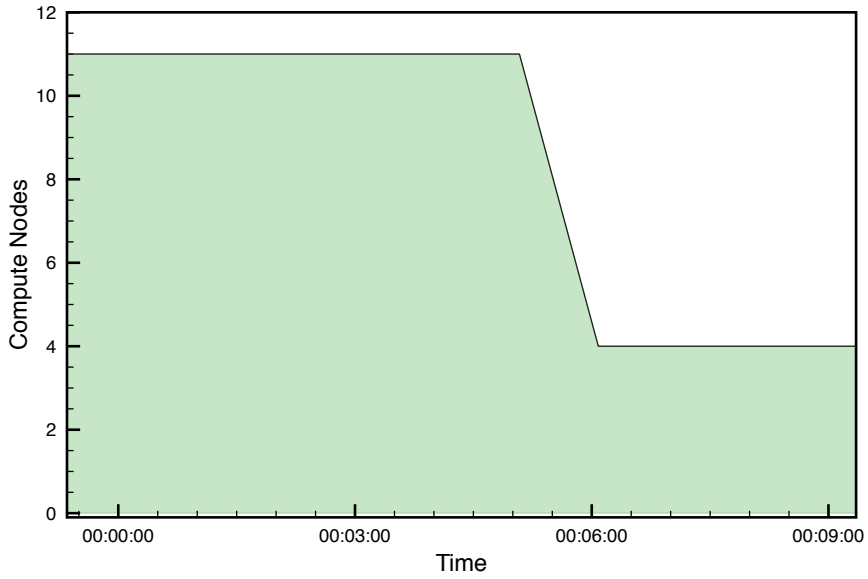

Figure 6.2: *Data showing the number of running compute nodes in the first run of algorithm 1.*

Figure 6.2 illustrates the difference between the environment before and after the policy according to the number of running compute nodes.

Experiment E_2 was successful and the test could without any exacerbation of the performance shutdown seven compute nodes. This is equal to save 63,64 % power consumption just from the compute nodes. The quality of service is not modified and all virtual machines in the environment have the same amount of resources as before the policy. Even with the delay caused by the single failure with one virtual machine the policy proofs the concept and there is no doubt that this is an interesting and new way to optimize the power consumption without quality of service restraints.

There are obvious shortcomings with this algorithm, as it may create gaps in some of the compute nodes as it only considers one and one VM. Also, it does not take the current placement much in account. For instance, if C_N happened to have the most virtual machines and C_1 the least, it would be better to go the other way in order to reduce the number of live migrations. Lastly, and most importantly, all virtual machines will be treated equal and will be crammed into as few compute nodes as possible. How hard this packing happens can be adjusted to be optimized

for power consumption or performance. This may severely affect the performance of all the virtual machines if the VCPU affinity is set to be less than one-to-one. One way to mitigate this would be to use very low constraints, like letting C_{n_VCPU} be equal or less of the physical CPUs of the compute node. However, this could lead to very sparse provisioning and little gain in resource savings. After all, a cloud is intended to do some overbooking of resources, but fine-tuning the constraints will be a challenge.

6.3 The experiments for policy2

Policy1 was a success but the algorithm do have shortcomings in regard to number of live migrations and booking of resources. These two constraints will be the strength of algorithm 2. This algorithm will intentionally, as earlier described take both the number of live migrations and the handling of CPU affinity into consideration when calculating the available space at each compute node. This is the experiments:

- E_3 Simulate CPU affinity by script for all machines located at a single compute node
- E_4 Simulate to check constraints regarding number of live migrations and sorting of compute nodes
- E_5 Simulate a execution for proof of concept for the algorithm and calculations
- E_6 Run policy 2 with 1:1 for important machines and 1:1 for not important

The main advantages of policy 2 is CPU affinity and number of live migrations, and this is also the two first experiments. To make E_3 possible every compute node has to split its CPUs into two sets. One set for high quality and one for low quality. Due to the ease of use the script controlling CPU affinity was moved out of the policy main script. This allows CPU affinity to be controlled after or before the policy. The also made it possible to easily customize the CPU sets and even do partial CPU affinity to a few virtual machines located at a compute node. E_3 was completed successfully within seconds and with no failures. The results from the virsh commands was written to screen which made it easy to see the crosspoints and the effect this could have in regard to performance. Each machine printed the CPUs it had access to which gave a clear overview.

E_4 is important feature for the algorithm and an complicated experiment. This experiment had to calculate the number of running machines and the combinations for minimizing the number of live migrations. The assumptions done in the implementation worked as intended and the final results in E_5 and E_6 will describe this in more detail.

Experiment E_5 was a simulation of E_6 which is required before doing a major change in a working environment. The purpose of E_5 is to prove that the concept is

working properly and that the right changes and commands is made. E_5 revealed a fault in the calculations which could have been critical if this was not a simulation. The calculation was corrected and the simulation was run again. This time with no errors.

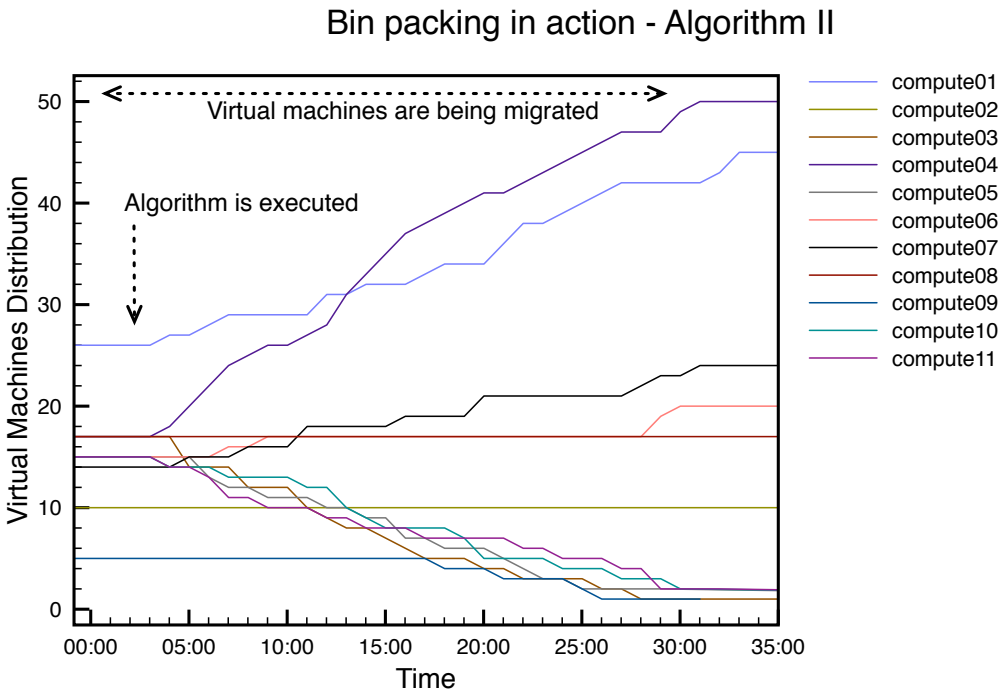

Figure 6.3: *This graph illustrates the number of virtual machines per compute node when the algorithm was started*

Figure 6.3 represents experiment E_6 and each line in the figure illustrates the number of running virtual machines located at the current compute node. In total there were 175 virtual machines in the environment. The figure shows that the environment is in a stable condition were the load of the system is divided among all compute nodes. Compute01 has around 25 virtual machines and had most machines. The rest of the compute nodes has between 10 and 20 virtual machines without compute10 who only had 6 machines. The algorithm was started four minutes into the graph and started out by calculating the compute node with the highest amount of virtual machines like this:

$$U^t_{VM_{rJ}} = \sum_{c=1}^{n} VM^t_i \tag{6.1}$$

When the right order of compute nodes is sorted out the algorithm starts to calculate the placement of virtual machines and stores the new location in the hash as described earlier in this chapter. As soon as all virtual machines has been assigned a new compute node or ordered to stay at the current location. The algorithm

calculated if there were space enough at the compute node by calculating the threshold like this for an important virtual machine.

$$\sum_{x=1}^{i} HQVCPU_x : HQVCPU_t \in C_n + VCPU_{vm} <= T_{HQVCPU^1} \tag{6.2}$$

If this equation is true the virtual machine is moved to computeX and the location is updated for later live migration. For Ã¥ virtual machine which is unimportant this is the equation:

$$\sum_{x=1}^{i} LQVCPU_x : LQVCPU_t \in C_n + VCPU_{vm} <= T_{LQVCPU^1} \tag{6.3}$$

The same goes for the unimportant virtual machine, if there is enough space at the physical machine the location is updated. When the algorithm has optimized the environment and all locations have been updated the live migrations took place. This started after 4 minutes and lasted 34 minutes.

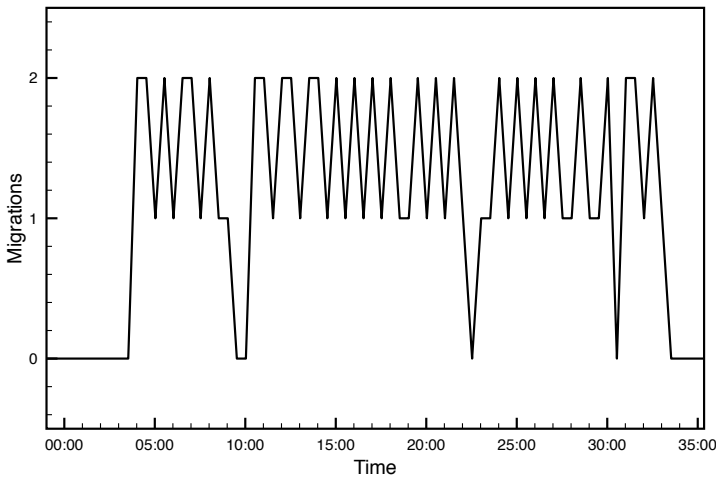

Figure 6.4: *A illustration of the live migrations during policy 2*

Figure 6.4 illustrates the live migrations in the environment. The small drops that happened at minute 10, 22 and 31 is migration of three larger virtual machines with more memory and VCPU than other machines.

Experiment E_6 was highly successful without any live migration failure and the policy was finished after 30 minutes. The policy started with 11 compute nodes and ended up with 6 compute nodes illustrated in figure 6.5.

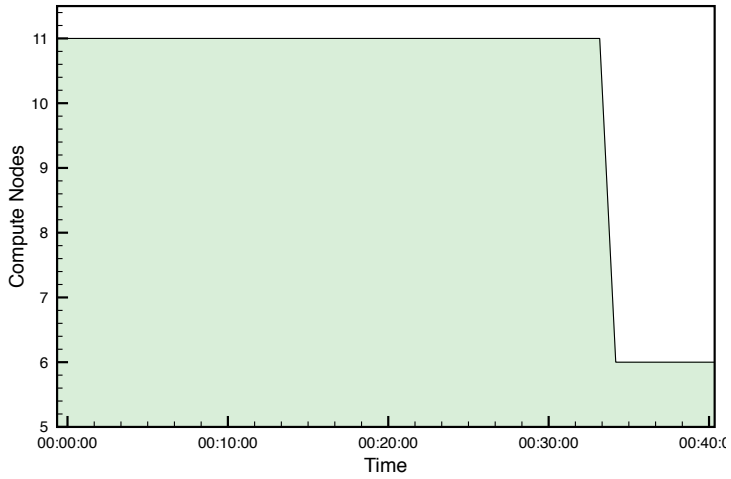

Figure 6.5: *Number of running compute nodes when policy 2 is applied*

6.4 The experiments for policy3

Both policy 1 and policy 2 is written specific for moving virtual machines in a cloud environment. The reason for trying a three dimensional bin packing algorithm is to compare the results in regard to calculation time, number of live migration and the how it would be possible to divide the machines into sets of important and unimportant machines. The environment consists of 198 virtual machines.

- E_7 Collect information and do calculations to input in algorithm
- E_8 Simulate a execution for proof of concept for the algorithm and calculations
- E_9 Calculate the number of live migrations and ensure the correct response

When experiment E_7 was conducted it became clear that the dimension of the bin could not possibly be the dimension of VCPUxMEMxIMPORTANCE because this created a bin so large that it would have room for thousands of virtual machines. The volume of the bin would have been 64x256x10. For all simulations run by E_8 all virtual machines fit into the first bin without any problem no matter how many virtual machines created.

In these experiments the important virtual machines where assigned weight 2 and non important virtual machines the weight 1. The point of setting the importance as the third dimension is to restrict the number of important virtual machines co-existing in the same physical machines. This means that one important virtual machines would cover the entire wall in the bin. This is explained in figure 4.15. The dimensions for the bins in policy 3 was 8x8x6, to best reflect the capacity wanted. This capacity would fit 24 important or 48 non-important virtual machines of the

size 2 VCPU, 4 GB memory.

Experiment E_7 was done several times to find the a optimal way to calculate the size of the virtual machines to make it understandable for the algorithm simultaneously not take virtual machines out of their proportions. All virtual machines where calculated by this example. A virtual machine having 2 VCPUs and 4 GB memory would correspond to a VM being a cuboid of the size 2(VCPU)x4(Memory)x2(importance). This means that the machine would take up the volume of 16. Experiment E_7 clearly indicates that this algorithm generate a significant amount of live migrations for every single run of the algorithm. This policy can end up with moving all virtual machines for every run. Experiment E_8 is decided to be done as a simulation.

The proof of concept was confirmed with the experiment E_8. The simulation used the file containing information of all virtual machines and the algorithm computed new locations to optimize the space at each compute node.

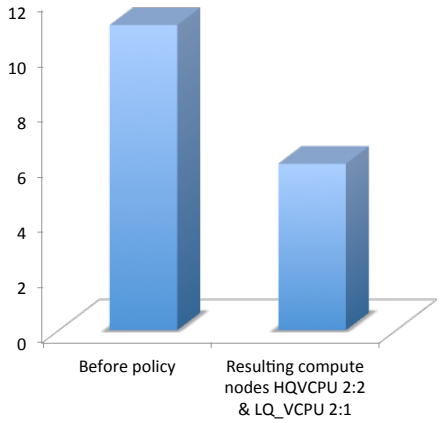

Figure 6.6: *This is a comparison of the resulted number of running compute nodes when running policy 3*

Policy 3 took the environment from 11 to 6 compute nodes which is a power consumption decrease of 45%. Experiment E_8 was a success as illustrated in figure 6.6 and the policy did pack the environment more optimized according to energy consumption. The drawback of this algorithm is the number of live migrations. As presented in figure 6.7 the number of live migraions is very high. The number of virtual machines in the environment when the experiment was conducted was 198 and the number of live migrations is 189. This is a critical high amount of live migrations.

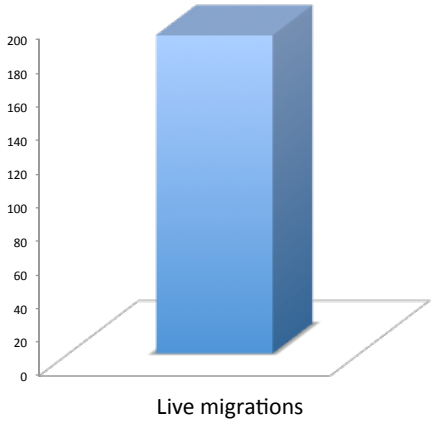

Figure 6.7: *A graph showing the number of live migrations during Policy 3*

6.5 Comparing the three algorithms

Now as the prototype works as intended with all three algorithms, the simulations for finding the best or most optimal algorithm can start. These simulations where conducted to find the best algorithm according to performance, live migration and energy savings. All simulations was conducted in the exact same environment for best comparison. To test the effect and optimization of the policy the test environment was upgraded with more virtual machines. There was created additional 75 virtual machines to get a realistic environment. Total amount of virtual machines where 198. All policies where asked to leave a buffer of one empty compute node at all times.

Each policy was simulated with two conditions regarding the density the virtual machines where packed at the compute nodes. The first simulation of policy 1 is described as S_1^{P1} and the second one as S_2^{P1}. Simulations of policy 2 and 3 will be referred to as S_1^{P2} and S_1^{P3}. The only constraint changing under the simulations was VCPU. OpenStack can overbook the VCPU up to 16 which means that 16 virtual machine CPU can be pinned to 1 physical CPU. By overbooking, the virtual machines can be packed harder which means that there is room for more virtual machines for every physical machine. The only exception is for policy 3 where the importance controls the space each virtual machines use. These simulations will determine which of the algorithms which are the best suited for the long term test.

- S_1^{P1} Simulate run of policy 1 with 1:1 vcpu
- S_2^{P1} Simulate run of policy 1 with 16:1 vcpu

- S_3^{P2} Simulate policy 2 with 1:1 for important machines and 16:1 for not important
- S_4^{P2} Simulate policy 2 with 1:8 for important machines and 16:1 for not important
- S_5^{P3} Simulate policy 3 with 2:2 for important machines and 2:1 for not important
- S_6^{P3} Simulate policy 3 with 3:3 for important machines and 3:1 for not important

Non of the migrations and shutdowns that follows was conducted in the environment due to the fact that the prototype is working and that there is no need to live migrate virtual machines and shutdown physical machines to get statistics already available. The policy was tested repeatedly in the experiment section and these statistics is to optimization. The result will be illustrated by graphs before a deeper analysis. Each group of bars in figure 6.8 represents one policy. One bar for each run of the policy with different density of the packing. The density is noted in the column below the bar. The bars going below zero is the number of live migrations. These numbers are divided by 10 to better readability.

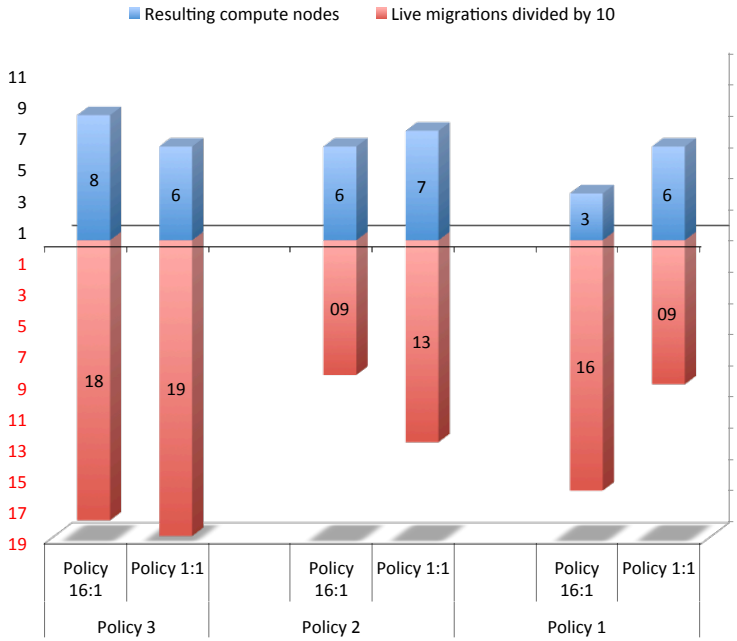

Figure 6.8: *A comparison of the number of resulting compute nodes and live migrations for all three algorithms. Live migrations divided by 10.*

By looking at the figure 6.9 there is one winner in terms of power reduction. Policy 1 with a VCPU density of 16:1 decreased the power consumption with 72%. The same policy saved 45 % power consumption without overbooking any VCPU. Let us now look closer at each simulation.

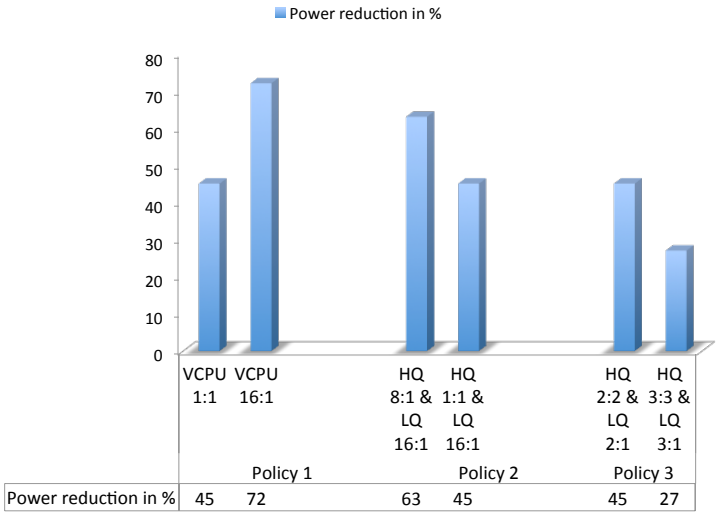

Figure 6.9: *A comparison of the power reduction for all three algorithms*

6.5.1 Policy 1

Policy 1 is the first proof of concept and is by far the simplest algorithm. This however is not the same as if the algorithm is the one performing poorest. The result for the algorithm was as follow.

Algorithm tested:	**Algorithm I**	
	$C_{n\_VCPU} = 1:1$	$C_{n\_VCPU} = 16:1$
Number of virtual machines	198	198
Resulting Compute nodes	6	3
Live Migrations	92	160
Power consumption reduction	45%	72%

Table 6.1: *Algorithm I tested on the same placement of virtual machines. A total of 198 virtual machines were running.*

This algorithm could without any form of quality of service reduction reduce the power consumption by 45% with 92 live migrations. This would leave the environment after the policy with six running physical machines. If the policy is allowed to overbook the VCPU, this algorithm reduce the power consumption with astoundingly 72 %. This was accomplished by 160 live migration leaving the environment only running 3 compute nodes where one physical machine is totally empty.

6.5.2 Policy 2

The conditions for the second policy is the exact same as for the first policy which is one of the benefits of just simulating. This algorithm is a more complex one which makes constraints like number of live migrations and makes different levels

of importance as described in earlier chapters. A closer inspection of the results is presented in the table below.

Algorithm tested:	**Algorithm II**	
	$C_{n\_HQVCPU} = 1 : 1$ $C_{n\_LQVCPU} = 16 : 1$	$C_{n\_HQVCPU} = 8 : 1$ $C_{n\_LQVCPU} = 16 : 1$
Number of virtual machines	198	198
Resulting Compute nodes	6	4
Live Migrations	86	129
Power consumption reduction	45%	63%

Table 6.2: *Algorithm II tested on the same placement of virtual machines. A total of 198 virtual machines were running.*

75 of the virtual machines where marked as not important which made it possible for the algorithm to overbook the VCPU. Simulation S_1^{P3} was conducted with one to one VCPU for important virtual machines and 16 to 1 for not important virtual machines. S_1^{P2} is the simulation with the lowest number of live migrations. With only 86 live migrations the algorithm could decrease power consumption with 45%. This is the same amount as S_1^{P1} which indicates that policy 2 could achieve the same amount of power savings with fewer live migrations. S_2^{P2} was conducted with the same density in regards to not important virtual machines but higher density for important virtual machines. These machines had one to one i S_1^{P2} but in this simulation they had 8-to-1. Every important VCPU shared one physical VCPU with 8 other VCPU. The performance will in this case possibly be lower that for S_1^{P3}. The simulation S_2^{P2} decreased the power consumption with 63% with 129 live migrations running the entire environment on 4 compute nodes. Of these compute nodes one was empty and working as a buffer.

6.5.3 Policy 3

The simulation S_5^{P3} reduced the power consumption by 45% by reducing the number of running physical machines from 11 to 6. The obvious drawback is the live migrations. S_5^{P3} made 189 live migrations which is the highest number of all policies. The reason for all live migrations is clearly the lack of the constraint that takes previous locations for the virtual machines into consideration. In simulation S_6^{P3} the difference between important and non-important virtual machines were higher as the non-virtual machines was only one third of the important ones. This also increased the size of the important one which made the power reduction only save 27% with 179 live migrations.
Consider that in a balanced environment, we would normally have about the same amount of virtual machines on every compute node, especially if they are similar in resource usage. It is therefore normal that algorithm I and II should perform almost equal in terms of the number of migrations as it does not matter where we start migrating from. The difference is that algorithm II separates important from not-important virtual machines providing a guarantee that too many important virtual machines randomly get packed on the same compute node. By studying

Algorithm tested:	**Algorithm III**	
	$C_{n\_HQVCPU} = 2 : 2$ $C_{n\_LQVCPU} = 2:1$	$C_{n\_HQVCPU} = 3 : 3$ $C_{n\_LQVCPU} = 3:1$
Number of virtual machines	198	198
Resulting Compute nodes	6	8
Live Migrations	189	179
Power consumption reduction	45%	27%

Table 6.3: *Algorithm III tested on the same placement of virtual machines. A total of 198 virtual machines were running.*

the results from the simulations there is one algorithm best suited for long term testing. By comparing the three algorithms, the best algorithm for long term testing is algorithm 2 with 1:1 for important and 4:1 for non-important virtual machines will be tested for three weeks .

6.6 Power consumption of a fully packed compute node

For the prototype to be able to save electricity there is obviously that the machines which is going down moves all the load from the present compute node to another. A interesting and important factor in this power saving is the fact that the receiving compute node do not use the double amount of electricity which would have critical for the power saving statistics. The phenomenon observed is not as expected, and is much lower than the hypothesis. In the figure 6.10 the policy was enabled saturday 26 and the machine went from a couple of machines to fully booked. It is possible to see that machine is having more spikes by looking at graph to the left. The graph to the right is the average of the last day and it is clear that the average is higher when the machine fully booked with virtual machines but it is very little.

The table below highlights the difference between compute node 05 when it went from a low number of virtual machines to be fully booked with virtual machines. The increase of power consumption went from 278 to 283 which is a increase of 1.80%. This illustrates the need to optimize the electricity efficiency and that it is possible to optimize the packing without increase of power.

Power consumption:	**Compute 5**	
	Low amount of VM‘s	High amount of VM‘s
Last day average consumption	278	283

Table 6.4: *Measurements of power consumption at compute05 when it went from low number of virtual machines to fully packed.*

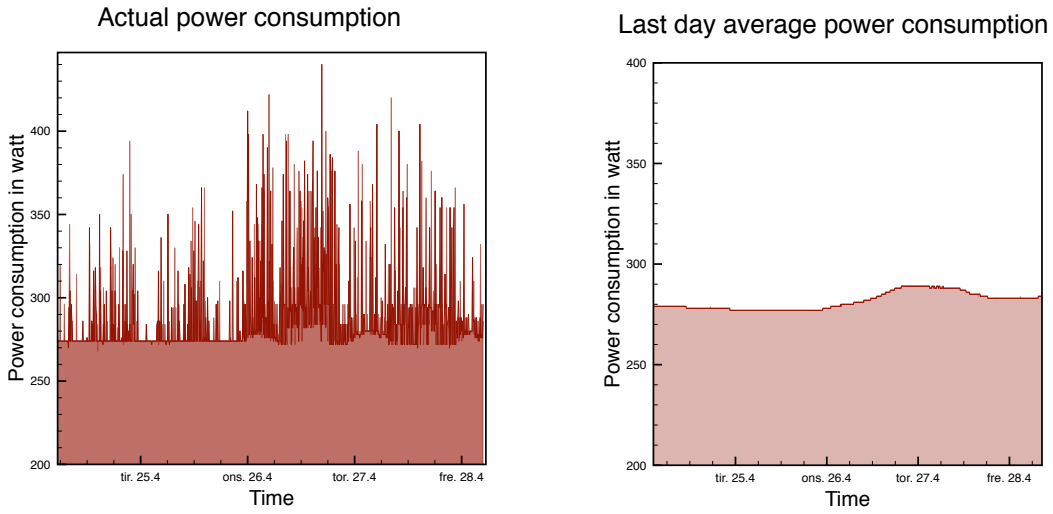

Figure 6.10: *Graph illustrating the power consumption when a compute node goes from a few virtual machines to fully packed.*

6.7 Temperature changes of a fully packed compute node

The temperature in the environment can both increase and decrease the power consumption of the environment by requiring more cooling. That is why it is important to see if the compute nodes that are being fully packed with virtual machines do not get a significant higher temperature than compute nodes in a balanced environment. The graph6.11 is an illustration of the temperatures in the environment during an execution of the policy. The figure to the left is from a compute node going from only a few virtual machines to be fully packed. As one can see there is a slight increase in the temperature, during live migration. The burst of virtual machines where created one this physical machines which is the peek in temperature by some degrees. The compute node was emptied for virtual machines during the night. In the figure to the right the total temperature is listed. This is not the degrees in the environment but all compute nodes put together. So there is not over 300 degrees but all machines together. As one can see the degrees significantly drops when the policy is executed but this is natural since the policy shuts down half of the environment.

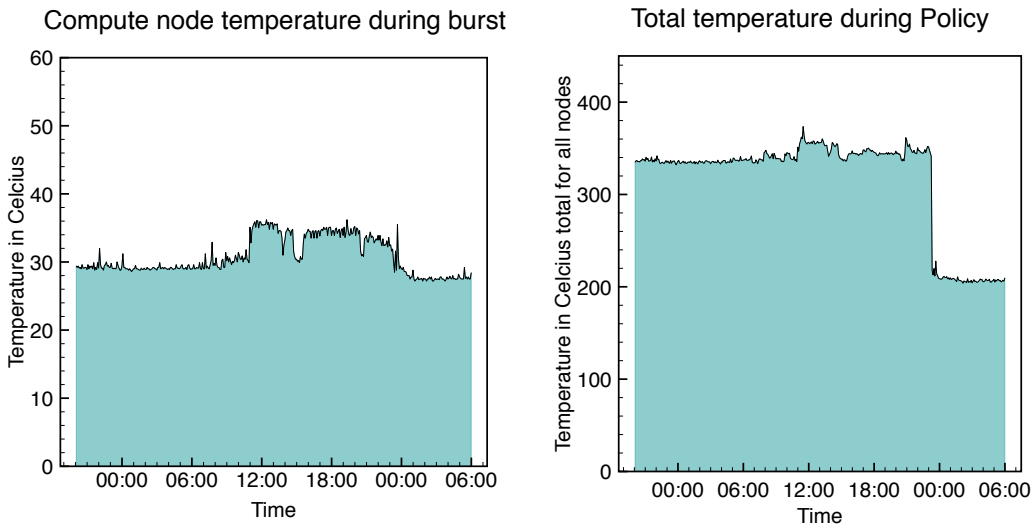

Figure 6.11: *Graph illustrating the temperature when a compute node goes from a few virtual machines to fully packed, and a graph of total temperature of all compute nodes*

6.8 Long term testing

The long term testing was conducted in the same environment as the simulations and experiments. The policy chosen for the long term testing was policy 2 due to the possibilities to lower the amount of live migrations and the ability to determine the difference between important and not important virtual machines. There is mostly important virtual machines in the environment and all virtual machines are given one-to-one VCPU. The environment is filled with virtual machines doing experiments for other master thesis so there is no overbooking. When the policy was started there where around 80 virtual machines, but within the first week the number of virtual machines raised to 150. The policy will be executed every hour, every day all week for three weeks. The long term policy takes the number of spare physical machines as an argument which allows the padding to be different according to night and day. In the design phase of this project there where suggested that the threshold and buffer should be calculated in percent like this where T_{VCPU} is the available space:

$$1 - \frac{U^t_{VCPU}}{C_{tn}} > T_{VCPU} \tag{6.4}$$

This would make it possible to have for instance 40% buffer of free hardware. For this long term test the buffer will not be a percentage but number of idle machines. This means that there will always be a given number of empty compute nodes running. There will be 1 compute node as padding during the night while it though daytime will be 2 compute nodes running as padding but without any virtual machines.

Before the policy was started the environment was i a balanced state which means that all compute nodes where running and had lots of idle hardware. This generates lots of live migrations at the first execution. The graphs 6.18 illustrates the starting of the policy.

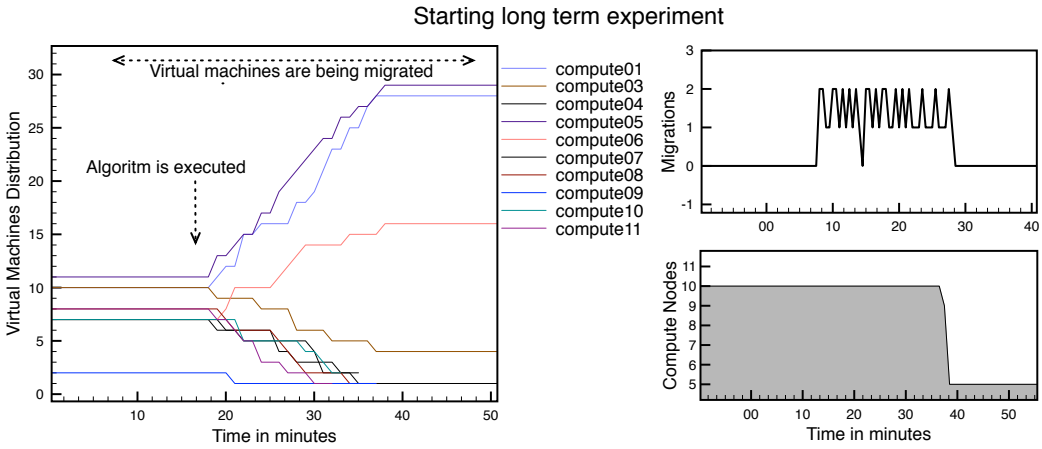

Figure 6.12: *Graph illustrating the startphase of the long term test of policy 2, all compute nodes to the left, live migrations top right and compute nodes active bottom right*

The policy first started to calculate new locations to optimize the space available at the nodes which are sorted according to number of virtual machines. After all calculations is done the live migrations of the virtual machines is conducted. These are illustrated at the top right graph. It took about 20 minutes to live migrate all virtual machines to new locations. After all virtual machines had gotten new locations the policy took down 5 compute nodes.

Compute05 is one of the compute nodes which received lots of virtual machines and it could be interesting to see the power consumption when it received 20 virtual machines. In figure 6.13 the power consumption of compute05 is displayed to see what happens when a physical server receives several virtual machines. The power consumption at compute05 spikes the minutes it is receiving virtual machines. The server consumed 100 watt more when receiving virtual machines but after a couple of minutes it goes back to normal again. The graph to the right illustrates the total consumption the environment while the prototype executed for the first time. It clearly shows a significant decrease of the consumption when 5 compute nodes where shutdown.

The two graphs 6.14 illustrates the number of running compute nodes for the first 11 days of the prototype. The policy was set to have 2 machines as padding during the day to handle new bursts of virtual machines and 1 during night. from 06:00 to 18:00 there where 2 in padding while it from 18:00-06:00 was one extra machines. The graph to the right also have a grey field which indicated the original state of

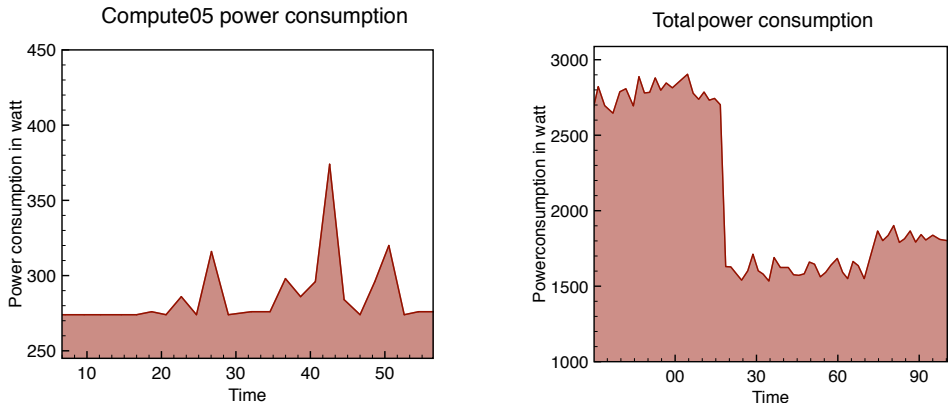

Figure 6.13: *Graph illustrating the power consumption of compute05 when it receives virtual machines and the total consumption of the environment*

the system if the policy was not running.

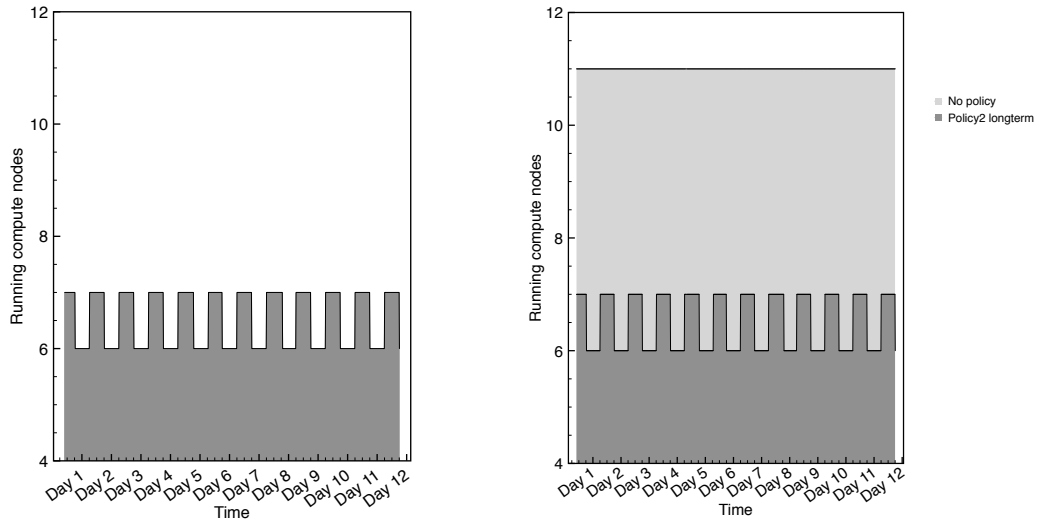

Figure 6.14: *Graph illustrating the number of running compute nodes the first 11 days of long term test, and the same graph which combines original state and prototype state*

After the first execution of the prototype there rest of the executions are left doing only little adjustment to the environment. This is also the reason for choosing policy 2 and not policy 3. Policy 3 would constantly moving virtual machines for every run. The lines in the next graph 6.15 is the number of virtual machines located at each compute node. As one can see there is not much movements in the environment only smaller adjustments after different virtual machines have been created.

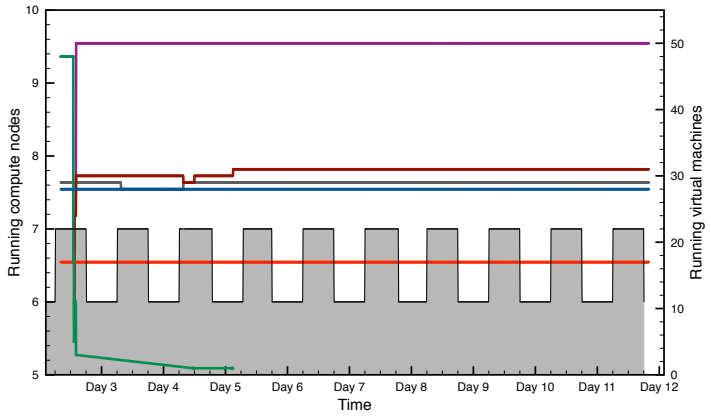

Figure 6.15: *Graph illustrating the number of running nodes with Y axis to the left and the number of virtual machines during long term test with Y axis to the left*

To illustrate how small the adjustments the prototype does after the first run, the graph 6.16 displays the live migrations done by the prototype.

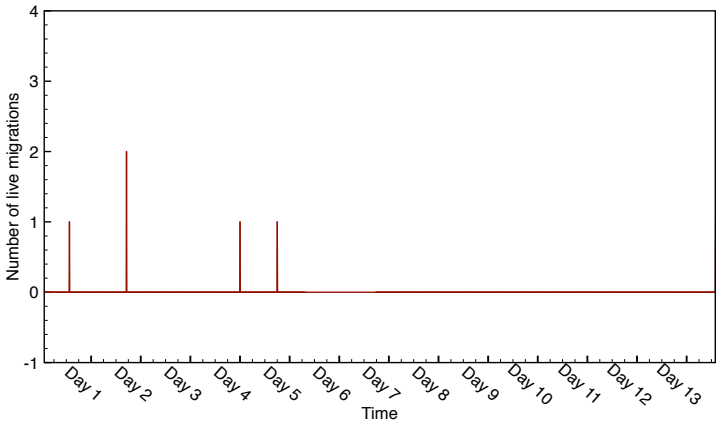

Figure 6.16: *Graph illustrating the number of live migrations during the long term test*

To take a closer look at the actual power consumption and what the policy saved the number is presented in the table below.

Power consumption:	Policy 2 long term	No policy
Watt usage 1 day	43 992	74 448
3 weeks usage	923 kwh	1563 kwh
1 year estimated usage	16 057 kwh	27 173 kwh
Power consumption reduction	40%	0%

Table 6.5: *Long term test of policy 2. Three weeks with policy 2, 2 spare machines at daytime and 1 in the night. No performance constraints.*

The policy reduced the number of running compute nodes from 11 to 6 at night and 7 during the daytime. There was two empty compute nodes during the day to have hardware available for new virtual machines created by any users. This decreased the electricity with 30 000 watt per day. In the three weeks of testing the policy saved 630 000 watt. This equals a power reduction of 40%.

6.9 Burst of virtual machines

The algorithm was executed every hour in the long term test. This would theoretically mean that it could happen that the performance of all virtual machine can be lowered if somebody created and started a burst of multiple new virtual machines. This happened several times during the long term test. Lets take a closer look at one of the bursts.

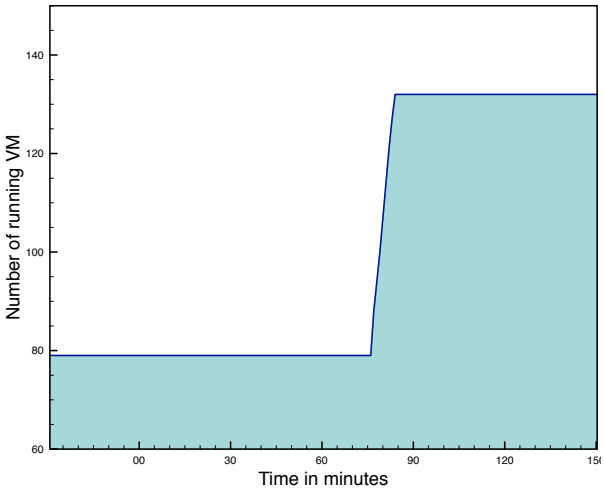

Figure 6.17: *Graph illustrates the number of running virtual machines in the environment during the burst.*

The number of running machines went from 79 to 132. The increase of 53 virtual machines equals a 67% boost in the environment. By looking at the number of free

VCPU cores at each running compute node we can see the impact this burst had to the performance.

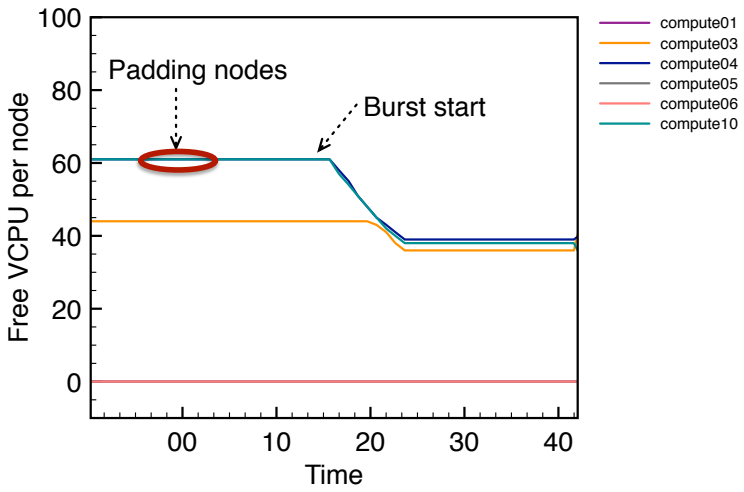

Figure 6.18: *Number of free VCPU cores per compute node during burst and policy execution.*

The burst that almost doubled the number of virtual machines happened during the day when there where 2 empty compute nodes as padding. In the graph both compute10 and compute04 have 64 available cores which means that the physical machine is running but there are no virtual machines located at the compute node. When the boost of virtual machines happened these two compute nodes got reduced to 39 and 38 free VCPU while compute03 only got decreased to 36. So the almost doubling of the number of running virtual machines did lower the amount of free hardware but the environment still had 113 cores left, until next policy was executed.

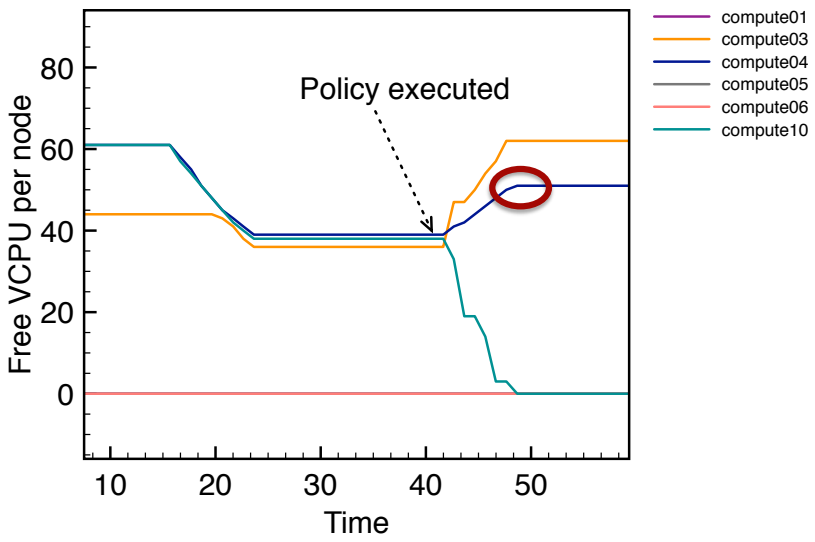

Figure 6.19: *Graph illustrates the number of free VCPU cores during the burst of virtual machines, and the following execution of the policy.*

When the scheduled algorithm was executed the policy live migrated virtual machines from compute03 and compute04 to compute10. This was done to pack the virtual machines optimally, and to try to empty two compute nodes so they could be used for padding. Compute03 was cleared and could be used as padding. As one can be seen in the red circle in figure 6.19, compute04 was not able to remove all the virtual machines located at the compute node. This triggers the policy to start a new compute node as can be seen in figure 6.20.

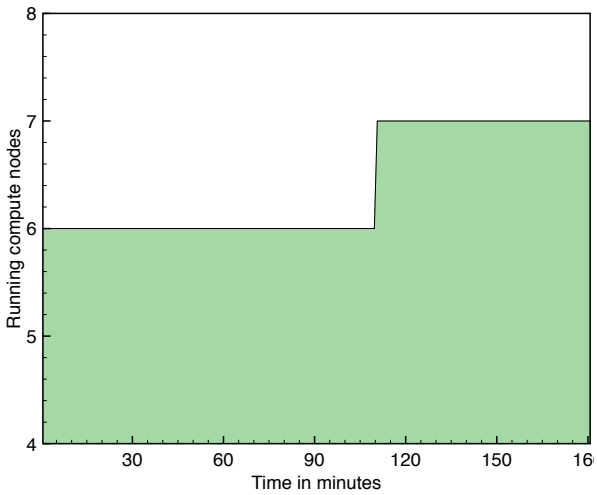

Figure 6.20: *This is a figure showing the number of running compute nodes during the execution of the policy.*

Chapter 7

Discussion: Managing a scaling cloud

From the variance in approaches found in similar research, there are several ways to design and implement this model. The approach chosen for the CERES project was based on the experience previous work had when trying to reduce energy footprints of cloud computing. The choices made in the process of this thesis will in this chapter be discussed. This includes the approach, design, implementation of the prototype and the results of the experiments.

7.1 The model

Some of the features discussed in the introduction and problem statement were energy efficiency, optimization and dynamic scaling. This thesis demonstrates and explains how energy efficiency can be optimized by dynamically scale the compute nodes in a cloud environment. The result and findings from the proof of concept and experiments illustrate that the prototype developed is able to optimize the use of energy. The results are based on a solid foundation of several thousands monitoring lines and the results give a significant reduction of power consumption for all three algorithms. The most important result is although the design and model of the prototype. The prototype is working and gives proof of concept that energy efficiency can be optimized by dynamic scaling of the compute nodes in a cloud environment. The analysis determined how and how much of the consumption can be reduced.

7.1.1 Implementation

One of the first things implemented in CERES was the monitoring module. This module was important for the project for two reasons. First in terms of building a machine learning software able to learn from the environment and to do calculated decisions based on this information gained from the environment. Secondly to retrieve empirical evidence when running simulations and experiments.

The amount of data retrieved from the environment every single day was 169 056 lines of information inserted into the database which had to be closely analyzed to see the influence the different policies had to the environment. The time spent building a precise and robust database would prove to be time well spent. The amount of data collected and analyzed would not be remotely possible to handle without opportunity to retrieve information based on a interval of time. The data was intended to describe power consumption which was done by collecting reliable data directly from the interface controlling the consumption on each compute node. Data regarding the OpenStack environment was retrieved from OpenStack's internal database with some precise queries which returned information accurate and properly.

One difference in the scripts as opposed to the pseudocode, was that virtual machines were not migrated directly, but scheduled for migration. This allowed the resulting migrations to be reviewed first for safety. Once the scripts were tested sufficiently, the migrations were conducted at the end of the script automatically. The compute nodes were managed through a separate console interface, allowing collection of power usage and to power on compute nodes that were shut down. OpenStack did not provide an interface to classify important VMs and non-important ones. In order not to interfere with the tenants, the non important virtual machines were defined in a separate file so that by default any VM is important unless otherwise stated. Before every test, he current placement of each virtual machine was recorded so they could be set back to it's original place after each experiment. For a comparison, the placement of the virtual machines was simulated so all three algorithms with a variations in the parameters could be tested. A total of 198 virtual machines were active at the time of the experiments. The compute nodes were shut down after all migrations were complete. This, also, was for concerns that there might be a erroneous situation and all actions should be postponed until the desired state was verified.

7.2 The difference between the policies and comparison with other research projects

The first policy was able to reduce the power consumption by 45% without any performance constraints, and by packing virtual machines with overbooking calculations from OpenStack the policy saved 72% of the power, which made the environment go from 3 kwh to 0,8 kwh which is a significant reduction. Policy 2, with more constraints than policy 1, decreased the power consumption with 45% and 63% with different settings regarding classification of non-important virtual machines and lower amount of live migrations. Policy 3 did decrease the power with 45% and 27% but with a significant increase of live migrations. The results from the simulations and experiments confirms that it is possible to save power by making adjustments in a cloud which was experienced in earlier projects as VirtualPower [26], Vgreen [6] and DVFS [36]. These projects did get a reduction of the power consumption, and the Vgreen project [6] did reduce the power consumption with 40% percent without performance constraints. VirtualPower [26] project did

show a improvement of 34% in a heterogeneous server system. There are several project that confirms the assumptions and experiences made in the CERES project. It is possible to save power and the savings is in a significant amount. The CERES project although its preliminary stage, contributes by getting one step closer to finding a automated way to save power without manual interference of system administrators.

Algorithm I, although simplest, does provide the most predictability and control. Packing from the highest compute node to the lowest means that C_1 always is on while C_N most likely is off. If C_4 is on, C_3 should be on too. This makes it easier to manage the compute nodes. When using algorithm II, we had to develop special tools to figure out what compute nodes were running at any time. OpenStack itself was not modified in any way.
In policy 3 the volume of a virtual machine is the square root of the product of VCPU and memory. The volume of a bin or physical machine is 8 x 8 which would give a 64 slots to place virtual machines. The downside of the calculations which made it possible to use the third algorithm is the complexity. It would be hard to implement this in a working environment where a system administrator is required to do complex calculations to be able to use this way of solving the three dimensional bin packing problem. Another challenge one encountered while using this algorithm is that the algorithm could turn and rotate the boxes which would make it possible for a machine to change the direction in a box leading to that the VCPU becomes memory and opposite.

The prototype and gathered data is reproducible by reading the design of the prototype and by following the model. By studying the complete scripts in appendixes which are thoroughly commented this project can be reproduced. Be aware that the analysis is coherent to the environment and a larger change in for instance virtual machines will affect the results.

7.2.1 Long-term testing

Algorithm II was selected for long-term testing over 3 weeks as this algorithm had the most stable behavior in terms of migrations and allowed us to protect the other virtual machines. The long term policy was configured to run every hour keeping two physical machines as padding during day and one during the night. In this way the policy could handle burst of new virtual machines without performance constraints to the already running virtual machines. All virtual machines had one-to-one hardware, which means no performance constraints. The results of the long term test was a electricity reduction of 40%.

The policy dynamically reduced the number of running compute nodes without any manually interference from system administrators. The reduction was from 11 to 6 compute nodes during the night and 7 during the day. The amount of power saved is relative to the amount of virtual machines and the factors of over provisioning. By adjusting both the working hours and the number of spare physical machines running there are definitively more electricity to save without hurting the perform-

ance. This can in the future by done by calculating the threshold for padding in percentage like suggested in the design chapter.

Throughout the long term test there where several instances of bursts of virtual machines. These could generate a situation where the environment got overbooked by virtual machines and therefore hurting the performance. One instance increased the environment by 67% which lowered the number of available hardware but did not make any performance constraints. An interesting scenario could be to calculate the threshold of the environment by calculating the quotas in OpenStack. Each user in OpenStack has its own quotas in regard to the amount of virtual machine that user can create. This would be a nice way to control the padding in the environment. It could, for instance, say that during the night the quotas where lower than during the day to ensure quality of service. This could also control the padding during the day by letting users only create a certain amount of virtual machine every hour.

During all tests and experiments we faced some major considerations. The first challenge was the technical difficulty of combining several technologies to both interact and make changes in a working environment in production. This required a complex model of the prototype which would not interfere with other projects running on virtual machines. This also required that the experiments and simulations were well planned and well tested with zero fault tolerance before conducted in the environment. The last challenge was the amount of data required to say how the experiments and simulations went. All these take into account the need for a good plan was present. The ambitious plan and approach was followed throughout the project with only small changes according to the results and ideas gotten during experiments. As the project evolved there were many ideas for additional features that could be implemented in the prototype. Some were implemented and other will be presented in the section of future work. The project experienced several obstacles in form of technical challenges but was solved sequentially as a result of good support and long hours. By dividing the project into several periods it was easy to see if one was in front or behind the schedule, and therefor possible to adjust the working hours according to the plan. Below are the major considerations.

7.2.2 Avoiding live migrations

Algorithm I and II both result in many live migrations when executed on the cloud with all compute nodes running and virtual machines spread over all of them. However, subsequent iterations produce very little change if the number of virtual machines has not changed. However, the main difference between the first two algorithms and the third is that the third will potentially re-arrange all virtual machines every time the algorithm runs if there has been a small change in the number of virtual machines. This is a typical assumption for bin packing algorithms: they start off with empty bins. Based on the interval of how often the algorithm runs, this may amount to a high level of migration activity.

In some situations we noticed unsuccessful live migrations where virtual machines

were migrated yet OpenStack would not register it properly. Live migration was therefore something we tried to avoid as much as possible. For our production environment it was considered unsafe to run algorithm III over a long period for testing. Further tests in a lab environment are necessary to document this behavior and to investigate the potential hazards of multiple live migrations at every interval.

7.2.3 Target nodes and power consumption

By comparing the actual power consumption of a compute node with a small amount of virtual machines and the fully packed compute nodes it is possible to see that the consumption is almost the same. The electricity consumption spikes when the compute nodes are receiving the virtual machine but except the live migration the consumption of electricity of a full compute node is equal. This proves the theory proposed by Duy et al. which estimates the power consumption to be a linear function of CPU, memory and disk usage [7]. This is explained in section 2.6.1. So as long as the virtual machines are idle there will be no increase in power consumption. Beloglazov et al. also states that the consumption of a idle physical machine is 70% of the power consumed when it is fully utilized [2]. This indicates that the consumption will only increase 30% from idle to fully utilized proving that the prototype will save even more power if virtual machines are running hardware demanding activity.

7.2.4 Adding extra capacity

One interesting aspect of our research is what happens when tenants create more virtual machines and there are no vacant compute nodes around. Booting up a compute node is not part of the scheduler in OpenStack, so the launch action would simply fail. In order to cope with bursts of new virtual machines, we added a level of padding. During daytime the padding was 2 compute nodes extra and 1 during night time. This means that if the algorithm would pack all virtual machines in 3 compute nodes, 4 would stay up (5 during the day), providing extra room. This practical consideration comes at the cost of more power consumed. An interesting alternative would be to modify the OpenStack nova scheduler in order to boot up a compute node and wait for it to come up before scheduling the virtual machine. The time a compute node required from start to it was up and running was as little as 3 minutes which makes it possible to lower the padding if OpenStack could start a new machine.

7.2.5 Algorithm intervals

The interval at which the script is executed determines how reactive the cloud is to changes. In our tests we experimented with 1 hour intervals during the day and 2 hour intervals during the night. We found this to work well in the clouds workload in this period. A shorter interval would allow the cloud to grow faster in case of high demand. In a balanced normal environment the policy would not make any changes unless a major change in number of virtual machines, this goes for policy

1 and 2. This would make it possible to have shorter time interval between each run since the policy does not demand any resources.

7.2.6 Monitoring and other scheduled tasks

In terms of monitoring, we had to make sure the compute nodes were silenced in the monitoring system when they were shut down and re-enabled when booted up again. This poses an interesting problem that challenges standard monitoring systems where "down" inherently is an alert state and the only other state to choose from is “maintenance”, which would silence the alarms, but not be the correct description. Conversely, if a compute node was supposed to be shut down yet was still running, we might want this to be triggered as an event too.

In our case, Sensu [33] was used for monitoring, which enabled the insertion of special checks for the compute nodes that could cope with some of these situations. Sensu uses agents on the compute nodes. The agents announce themselves and are automatically added. This means that we only had to delete the compute node from sensu when it was shut down and it would automatically reappear when it was booted. Metrics were collected into OpenTSDB, which, with it’s tagged data, automatically added the new data to its graphs.

For monitoring in case there should be an issue with the algorithm itself, we also included checks to catch if a virtual machine was considered migrating for too long (30 minutes) and if a compute node was scheduled for shut down but still had remaining virtual machines.

Other scheduled tasks such as backups and configuration changes had to be taken into account as well. Using configuration management, compute nodes could be rebuilt without relying on a backup and it was therefore not considered essential to back up all the compute nodes all the time. There was no major change in the configuration during testing, but the plan was that all compute nodes would be booted manually prior to upgrades.

7.2.7 Levels of importance

The classification of what virtual machines are important is outside of this algorithm, but in our case we let it be something the tenant would choose. Combined with a future billing system that would take this into account, the user would have an incentive to classify unused virtual machines as not important for a period of time. One could also detect importance by looking for special flavors (e.g instance types in Amazon), but that would render a virtual machine important for it’s entire lifetime, which is impractical. On the other hand, machine learning approaches could attempt at this classification, like the learning model presented in the Dependable Virtual Machine Placement project [45].

7.2.8 Alternative approaches

Other approaches in performance optimization and energy saving modify the clouds deployment scheduler to decide where to put the VM when it is created. As an alternative to our approach, one could just start an empty cloud with one compute node and fill it completely until the next one is booted up. However, without live migration, scaling down may be an issue as compute nodes end up being underutilized and difficult to remove as long as they still have a few remaining virtual machines.
The approach chosen for this type project is a combination of all the projects listed in the background chapter. By studying earlier project with similar problem statements it became clear that the approach for this kind of project was crucial and the importance of a design and module phase was obvious. The common approach for this kind of project is without a doubt done with simulations. In retrospect the approach of the project could include more simulations and less implementation to save considerably amount of time, but the implementation is also one of the parts which makes this project interesting and unique. A unique feature for the prototype is the ability to use the same software but change the algorithm. The ability the software had to change the algorithm without reprogramming the entire program was crucial to be able to compare several algorithms. A change in the approach could have made changes to the approach but not considerably. A working prototype should either way be developed but the measurements and the number of algorithms would be changed. Both a case study and a benchmark could after this project be possible and is suggested as future work to even further optimize the algorithm. Research wise there were spent a considerably amount of time in the beginning studying other research before making an approach which could have been shortened down for more implementation time, but in retrospects, this was also a factor for making a successful prototype.

The CERES prototype assumes that the cloud may not be packed at all times, and that the administrator may want to enable power-saving only when desired. The ability to transition between power saving and workload optimizing strategies seems the optimal approach for us. Furthermore, with clouds using a release cycle of once to twice a year, this would mean having to update the schedulers code at every upgrade. CERES would only have to be modified in how the state information is collected and the command for live migrations in order to be adapted to a different cloud solution, allowing us to separate the management strategy from the technology.

The potential impact of the prototype and algorithms is a new way of controlling resources in a cloud environment. The efficiency of the power consumption can contribute to decreasing the power consumption of data center which we already now is very high. In the specific environment the prototype is tested it saved as much as 45% without lowering the performance which means there is no reason for not saving that power. The clouds are using the same amount of electricity as countries like Brazil and Germany [30] which means that there is a significant amount of electricity possible to save. This will also affect the operational cost which can contribute to the opportunity for lower cost of cloud services.

With policy 2 it would also be possible for users to say that a virtual machine is not important during the night which tells the environment that this machine can be packed harder with lower hardware demands during the night, and therefor save both the customer and the data center for operational cost.

The project have developed a new approach for managing clouds and several projects, problems and questions have emerged from the project. A new tool has been developed and a deeper insight into power saving options at data centers. This affects both end users, system administrators, cloud administrators and the economics regarding cost efficiency.

7.3 Future work

During the development and testing several research projects was discovered along with additional features for the prototype. To ease the user experience it would be great to implement a graphical interface for Horizon which is the interface for OpenStack users. This should also have new features like a new and better way of monitoring the different aspects of a scaling cloud. A useful new monitoring feature would be a surveillance of compute nodes running and stopped.

Future work will be to incorporate workload optimization algorithms and to enable the cloud to switch between strategies. Beside adding additional features to the software it is possible that CERES can optimize the environment not only for power optimization but also for performance. In this way it could easily be implemented an feature where the prototype can optimize according to power consumption at night while at day time optimize for performance. This is interesting because it is a win-win situation where the prototype not only saves money but also contribute to higher performance. This would be a combination of the project Btrplace [11] and this project.

The new monitoring features would make it easier and more intuitive to implement re-balancing capabilities for the prototype. This should be used to re-balance the environment when the policy is stopped or canceled. One interesting project for future work is to switch between different algorithms based on the time of day. The re-balancing was a feature longed for during all the experiments. Another monitoring feature longed for during the long term test is the ability to make alerts if a live migration takes longer time than usual or if if fails. This could be done in the prototype where it already checks that the compute node shutting down is totally empty. The prototype already collects, generates and stores a significant amount of information which could be turned into detailed reports of power consumption and savings, along with performance, number of virtual machines.

An interesting future feature for system administrator would be if the prototype also could configure itself without any interference. This would happen by collecting required information already stored in the database. Another self configuration feature should be the importance level provision. The virtual machines should have

the ability to decide if it is important or not based on the multiple criteria like hardware demand, users logged on the system, processes running and so on. This could lead to an even more dynamically cloud where all idle virtual machines are stored on as little hardware as possible leaving more optimized locations for important machines. By packing even harder it would generate more research for other types of constraints, like if two virtual machines should not be on the same compute node for high-availability and separation of IO loads.

A more advance padding features for lower power consumption where it is possible to allow padding to be 40% free hardware at one compute node instead of one separate physical machine. This would allow the system to save 10-20% more electricity. This could be combined with the ability to let OpenStack boot new physical machines if needed to lower the need for padding. The startup time for a physical machine was low enough to decrease the required padding. For system administrators a required feature would be an easy way to boot all physical machines for important updates and maintenance. This would also provide the ability to live migrate all virtual machines to ease maintenance procedures for administrators.

Chapter 8

Conclusion

The main goal of this project was to investigate how energy efficiency can be optimized by dynamic scaling of computer nodes in a cloud environment.

The problem statement is addressed by developing a prototype to demonstrate how energy efficiency can be optimized. CERES is built by using three different multi-dimensional bin packing algorithms implemented on a large Open Stack deployment.

Our results and experiences show that each algorithm has different effect on the balance between energy efficiency and performance. The simplest bin packing algorithm provides useful mechanism to reduce power consumption, while more complex algorithms do not give the same power saving, but give better performance.

The project was able to optimize the energy efficiency by lowering the energy consumption with 40% without reducing performance. An even higher reduction can be achieved by overbooking the hardware capacity. Some experiments did reduce the consumption with as much as 72%.

There are still some practical challenges to be addressed in future research;

- The adaption of the algorithm to scenarios in terms of defining the constraints
- Adjusting the management tools and approaches to cope with scaling hardware

An article about the CERES project has been submitted to the 28th edition USENIX‘s Large Installation System Administration conference with the title "Saving the planet with bin packing - Experiences using 2D and 3D bin packing of virtual machines for greener clouds".

Bibliography

[1] Kyrre Begnum. The mln project@ONLINE. http://mln.sourceforge.net/. Online accessed: 2014-03-04.

[2] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, page 4. ACM, 2010.

[3] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM.

[4] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. John Wiley & Sons, 2013.

[5] Hanen Chihi, Walid Chainbi, and Khaled Ghedira. An energy-efficient self-provisioning approach for cloud resources management. *ACM SIGOPS Operating Systems Review*, 47(3):2–9, 2013.

[6] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. vgreen: A system for energy-efficient management of virtual machines. *ACM Trans. Des. Autom. Electron. Syst.*, 16(1):6:1–6:27, November 2010.

[7] Truong Vinh Truong Duy, Yukinori Sato, and Yasushi Inoguchi. A prediction-based green scheduler for datacenters in clouds. *IEICE Transactions*, 94-D(9):1731–1741, 2011.

[8] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[9] IBM global. Virtualization in education @ONLINE. http://www-07.ibm.com/solutions/in/education/download/Virtualization%20in%20Education.pdf, October 2007. Accessed: 2014-02-10.

[10] Google. Data centers energy efficiency@ONLINE. http://www.google.com/about/datacenters/efficiency/index.html. Online accessed: 2014-02-17.

[11] F. Hermenier, J. Lawall, and G. Muller. Btrplace: A flexible consolidation manager for highly available applications. *Dependable and Secure Computing, IEEE Transactions on*, 10(5):273–286, Sept 2013.

[12] OpenStack Cinder homepage. Openstack storage@ONLINE. http://www.openstack.org/software/openstack-storage/. Online accessed: 2014-03-03.

[13] OpenStack Nova homepage. Openstack compute@ONLINE. http://www.openstack.org/software/openstack-compute/. Online accessed: 2014-03-03.

[14] OpenTSDB homepage. The scalable time series database@ONLINE. http://opentsdb.net/. Online accessed: 2014-03-04.

[15] VMware homepage. Virtualization@ONLINE. http://www.vmware.com/virtualization/. Online accessed: 2014-03-02.

[16] Damian Igbe. How to migrate an instance with zero downtime: Openstack live migration with kvm hypervisor and nfs shared storage @ONLINE, October.

[17] Businessinsider Julie Bort. The 10 most important companies in cloud computing@ONLINE. http://www.businessinsider.com/10-most-important-in-cloud-computing-2013-4?op=1, April 2013. Online accessed: 2014-02-15.

[18] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3):034008, 2008.

[19] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[20] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.

[21] Silvano Martello, David Pisinger, Daniele Vigo, Edgar Den Boef, and Jan Korst. Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Softw.*, 33(1), March 2007.

[22] Wolfram Mathworld. Np-problem@ONLINE. http://mathworld.wolfram.com/NP-Problem.html. Online accessed: 2014-03-02.

[23] Barry McQuarries. First fit bin packing problem@ONLINE. http://facultypages.morris.umn.edu/~mcquarrb/teachingarchive/M1001/Resources/BinPacking.html. Online accessed: 2014-02-25.

[24] Mayank Mishra and Anirudha Sahoo. On theory of vm placement: Anomalies in existing methodologies and their mitigation using a novel

vector based approach. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 275–282. IEEE, 2011.

[25] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 19–24, New York, NY, USA, 2009. ACM.

[26] Ripal Nathuji and Karsten Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. *SIGOPS Oper. Syst. Rev.*, 41(6):265–278, October 2007.

[27] Netgear. What is the jumbo frame supported by switches and adapters @ONLINE, May.

[28] OpenStack. Hypervisorsupportmatrix@ONLINE. https://wiki.openstack.org/wiki/HypervisorSupportMatrix. Online accessed: 2014-02-17.

[29] OpenStack. Openstack: The open source cloud operating system @ONLINE, February 2014.

[30] Green peace. Make it green: Cloud computing and its contribution to climate change @ONLINE. http://www.greenpeace.org/international/en/publications/reports/make-it-green-cloud-computing/, March 2010.

[31] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the electric bill for internet-scale systems. *SIGCOMM Comput. Commun. Rev.*, 39(4):123–134, August 2009.

[32] Margaret Rouse. Cloud computing@ONLINE. http://searchcloudcomputing.techtarget.com/definition/cloud-computing, December 2010. Online accessed: 2014-02-11.

[33] LLC(or) Sensu, Heavy Water Operations. The sensu monitoring system. http://www.sensuapp.org, apr 2014. Online accessed: 2014-03-02.

[34] The New York Times. The cloud factories power, pollution and the internet @ONLINE, September 2012.

[35] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52, January 2011.

[36] G. von Laszewski, Lizhe Wang, A.J. Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, Aug 2009.

[37] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing*, pages 254–265. Springer, 2009.

[38] Amazon EC2 webpage. Amazon elastic compute cloud@ONLINE. https://aws.amazon.com/ec2/. Online accessed: 2014-03-02.

[39] OpenStack webpage. Openstack neutron network@ONLINE. http://www.openstack.org/software/openstack-networking/. Online accessed: 2014-03-03.

[40] Wikipedia. Bin packing problem@ONLINE. http://en.wikipedia.org/wiki/Bin_packing_problem. Online accessed: 2014-02-25.

[41] Wikipedia. First fit bin packing problem@ONLINE. http://en.wikipedia.org/wiki/Bin_packing_problem#First-fit_algorithm. Online accessed: 2014-02-25.

[42] Wikipedia. Live migration@ONLINE. http://en.wikipedia.org/wiki/Live_migration. Online accessed: 2014-02-17.

[43] Wikipedia. Virtualization of hardware@ONLINE. http://en.wikipedia.org/wiki/Hardware_virtualization. Online accessed: 2014-02-10.

[44] Wikipedia. Virtualization @ONLINE. http://en.wikipedia.org/wiki/Virtualization. Accessed: 2014-02-10.

[45] H. Yanagisawa, T. Osogami, and R. Raymond. Dependable virtual machine allocation. In *INFOCOM, 2013 Proceedings IEEE*, pages 629–637, April 2013.

[46] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 145–154, New York, NY, USA, 2012. ACM.

[47] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th international conference on Autonomic computing*, pages 145–154. ACM, 2012.

[48] Zhaoyi Zhang, Songshan Guo, Wenbin Zhu, Wee-Chong Oon, and Andrew Lim. Space defragmentation heuristic for 2d and 3d bin packing problems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 699–704. AAAI Press, 2011.

Chapter 9

Appendix

9.1 Policy 1

Policy1.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables

my $VERBOSE = 0;
my $DEBUG = 0;
my $MIGRATIONPAUSE = 0;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname
my %status;
my %computestatus;
my %vms;
my %vms_old;
my %nametable;
my $maxrunningcompute = 0;
my $buffer = 0;
# command line options

my $opt_string = "vdh";
getopts("$opt_string", \my %opt) or usage() and exit(1);

$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

if($opt{'h'}){
	usage();
	exit 0;
}


##################################################
# Main part

getstatus();
getcomputestatus();
getservicestatus();

```

```
%vms_old = %vms;
printstatus();
printcompute();

# run algorithm which presents new status
my $crosspoint = agressiveRelocation();

printstatus();
runMigrations();
scaleservers($crosspoint);

# printcompute();


print "max running compute: $maxrunningcompute\n";


# print "\ncompute 02 free vmcp:
".$computestatus{"compute02"}{"compute_free_vcpu"}."\n";


################ Find one value in hash #############
# foreach my $computenode (sort(keys %computestatus)) {
#   #       print "$computenode: \n";
#       foreach my $item (keys %{$computestatus{$computenode}}) {
#       if ($computestatus{$computenode}{$item} == 55){
#           print "$computenode: $item: ".$computestatus{$computenode}{$item}."\n";
#       }
#   }
# }

################  Sort value after vcpu size ################
# foreach my $computenode (sort(keys %computestatus)) {
#   #       print "$computenode: \n";
#   foreach my $item (sort(keys %{$computestatus{$computenode}})) {
#       print "$computenode: $item: ".$computestatus{$computenode}{$item}."\n";
#   }
# }

################  Sort value after memory size ################
# foreach my $computenode (sort(keys %computestatus)) {
   #       print "$computenode: \n";
#   foreach my $item (sort(keys %{$computestatus{$computenode}})) {
#       if ($item == "compute_free_memory"){
#           print "$computenode: $item:".$computestatus{$computenode}{$item}."\n";
#       }
#  }
# }




################################################
# Subroutines
sub scaleservers {
my $crosspoint = $_[0];

   print "Crosspoint was at $crosspoint\n";
   # if crosspoint + buffer < maxrunning
   if( ($crosspoint + $buffer) < $maxrunningcompute){
      print "We are using too many compute nodes, scaling down\n";
      for ( my $i = ( $crosspoint + $buffer + 1 ); $i <= $maxrunningcompute; $i++ ){
         my $number = $i;
         $number = "0" . $number if $number < 10;
         my $computename = "compute$number";
         print "shutting down $computename\n";
      #    system("ssh $computename service nova-compute stop");
      }
```

```
114     # open(SSH,"ssh controller nova-manage service list | ");
115
116     }elsif(($crosspoint + $buffer) > $maxrunningcompute){
117         print "we are under capacity, scaling up\n";
118         my $number = $maxrunningcompute + 1;
119         $number = "0" . $number if $number < 10;
120         my $computename = "compute$number";
121         print "starting $computename\n";
122         # system("ssh $computename service nova-compute start");
123
124     }
125 }
126
127
128 sub getVMLocation {
129     my $vm = $_[0];
130     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
131 $password)
132       || die "Could not connect to database: $DBI::errstr";
133     my $sth = $dbh->prepare("select * from instances where uuid ='$vm';")
134       || die "$DBI::errstr";
135     $sth->execute();
136     my $results = $sth->fetchrow_hashref;
137     $sth->finish;
138     $dbh->disconnect;
139
140     return $results->{"host"};
141
142 }
143
144 sub getstatus {
145
146     # Get the rows from database
147     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
148 $password)
149       || die "Could not connect to database: $DBI::errstr";
150     my $sth = $dbh->prepare('select * from instances where vm_state="active";')
151       || die "$DBI::errstr";
152     $sth->execute();
153
154     # Print number of rows found
155     if ($sth->rows < 0) {
156         print "Sorry, no vm found failure in getstatus subrutine.\n";
157     } else {
158         # printf ">> Found something\n", $sth->rows;
159         # Loop if results found
160         while (my $results = $sth->fetchrow_hashref) {
161
162           my $vm_name = $results->{uuid}; # get the hostname
163           my $computenode = $results->{host}; # get the computenode
164           my $vm_vcpu = $results->{vcpus}; # VM number of cpu
165           my $vm_memory = $results->{memory_mb}; # VM number of memory
166           my $vm_project_id = $results->{project_id}; # VM number of memory
167            #   printf  $vm_name . " at ". $computenode . " with VCPU: " . $vm_vcpu .
168 " and memory: " . $vm_memory . "\n";
169
170            $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
171            $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
172            $status{$computenode}{$vm_name}{"project_id"}=$vm_project_id;
173
174            $vms{$results->{uuid}} = $computenode;
175            $nametable{$results->{uuid}} = $results->{"hostname"};
176
177         }
178
179 # Disconnect
180         $sth->finish;
181         $dbh->disconnect;
```

```
182     }
183 }
184
185 sub getcomputestatus {
186
187     # Get the rows from database
188     my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username,
189 $password)
190       || die "Could not connect to database: $DBI::errstr";
191     my $sth2 = $dbh2->prepare('select * from compute_nodes')
192       || die "$DBI::errstr";
193     $sth2->execute();
194
195 # Print number of rows found
196     if ($sth2->rows < 0) {
197         print "Sorry, no vm found failure in getstatus subrutine.\n";
198     } else {
199         # printf ">> Found something\n", $sth->rows;
200         # Loop if results found
201         while (my $results2 = $sth2->fetchrow_hashref) {
202
203
204             my $compute_name = $results2->{hypervisor_hostname}; # get the hostname
205             my $compute_vcpu = $results2->{vcpus}; # VM number of cpu
206             my $compute_memory = $results2->{memory_mb}; # VM number of memory
207             my $compute_vcpu_used = $results2->{vcpus_used}; # VM number of vcpu used
208             my $compute_memory_used = $results2->{memory_mb_used}; # VM number of vcpu
209 used
210             my $compute_running_vms = $results2->{running_vms}; # VM number of vcpu
211 used
212             my $compute_free_memory = $compute_memory - $compute_memory_used;
213             my $compute_free_vcpu = 64 - $compute_vcpu_used;
214
215
216             # print  $compute_name . " running VM: " . $compute_running_vms . " number
217 of used vcpu : " . $compute_vcpu_used . "free VCPU: " . $compute_free_vcpu  .
218 "\n";
219
220             $computestatus{$compute_name}{"running vms"}=$compute_running_vms;
221             $computestatus{$compute_name}{"compute_free_vcpu"}=$compute_free_vcpu;
222             $computestatus{$compute_name}{"compute_free_memory"}=$compute_free_memory;
223
224
225
226
227
228         }
229     }
230
231 # Disconnect
232     $sth2->finish;
233     $dbh2->disconnect;
234 }
235
236
237 sub printcompute {
238
239     foreach my $computenode (sort(keys %computestatus)) {
240         print "$computenode: \n";
241         foreach my $item (keys %{$computestatus{$computenode}}) {
242             print " $item: ".$computestatus{$computenode}{$item}."\n";
243
244         }
245     }
246 }
247
248
249
```

```
250 sub printstatus {
251     foreach my $computenode (keys %status) {
252         print "$computenode: \n";
253         foreach my $vm (keys %{$status{$computenode}}) {
254             print " vm-name:$vm\n";
255             foreach my $noe (keys %{$status{$computenode}{$vm}}) {
256                 print "  $noe: " . $status{$computenode}{$vm}->{$noe} . "\n";
257
258
259             }
260         }
261     }
262 }
263
264
265
266
267
268 sub getLoggingTime {
269
270     my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
271     my $nice_timestamp = sprintf ( "%04d.%02d.%02d:%02d:%02d:%02d",
272                                    $year+1900,$mon+1,$mday,$hour,$min,$sec);
273     return $nice_timestamp;
274 }
275
276
277 sub usage {
278
279         print "Usage:\n";
280         print "-h for help\n";
281         print "-v for verbose(more output)\n";
282         print "-d for debug(even more output)\n";
283
284 }
285
286 sub verbose {
287         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
288 }
289
290 sub debug {
291         print "DEBUG: " . $_[0] if ($DEBUG);
292 }
293
294
295 sub agressiveRelocation {
296     print "Agressive algorithm\n";
297
298     my $TARGET = 1;
299     my $VACANTER = $maxrunningcompute;
300
301     for ( my $i = $VACANTER; $i >= $TARGET; $i-- ){
302         my $number = $i;
303         $number = "0$number" if $number < 10;
304         my $computenode = "compute" . $number;
305
306         foreach my $vm ( keys %{$status{$computenode}}){
307             print "Testing for vm " . $nametable{$vm} ."\n";
308
309             for ( my $j = $TARGET; $j <= $i; $j++ ){
310                 $number = $j;
311                 $number = "0$number" if $number < 10;
312                 my $targetnode = "compute$number";
313
314                 print "trying to empty $computenode into $targetnode\n";
315                 if ( $i == $j ){
316                     print "Target and computenode are the same, we are finished\n";
317                     return $j;
```

```
318                 }
319
320                 my $cpu_ok = 0;
321                 my $mem_ok = 0;
322                 # target has enough vcpu
323                 if ( $computestatus{$targetnode}{"compute_free_vcpu"} >=
324 ($status{$computenode}{$vm}{"vcpu"})/ 16){
325                     $cpu_ok = 1;
326                 }
327
328             # target has enough memory
329                 if ( $computestatus{$targetnode}{"compute_free_memory"} >=
330 $status{$computenode}{$vm}{"memory"} ){
331                     $mem_ok = 1;
332                 }
333
334                 if ( $cpu_ok and $mem_ok ){
335                     print "VM will fit in, scheduling migration\n";
336
337                     # update %vms
338                     $vms{$vm} = $targetnode;
339
340                 # reduce compute_free_mem on target
341                     $computestatus{$targetnode}{"compute_free_memory"} =
342 $computestatus{$targetnode}{"compute_free_memory"} -
343 $status{$computenode}{$vm}{"memory"};
344
345                     # reduce compute_free_vcpu on target
346                     $computestatus{$targetnode}{"compute_free_vcpu"} =
347 ($computestatus{$targetnode}{"compute_free_vcpu"} -
348 ($status{$computenode}{$vm}{"vcpu"}/16));
349
350                     # add vm to $status{$target}
351                     $status{$targetnode}{$vm} = $status{$computenode}{$vm};
352
353                     # remove vm from $status{$computenode}
354                     delete $status{$computenode}{$vm};
355
356                     last;
357
358                 } else {
359                     print "VM did not fit in, assuming target is full\n";
360                     print "updating $TARGET to " . ( $TARGET + 1) . "\n";
361                     # $TARGET++;
362                 }
363
364             }
365         }
366     }
367
368 }
369
370 sub getservicestatus {
371
372     # nova-compute     compute09                       nova
373 enabled    :-)   2014-03-21 19:51:32
374     open(SSH,"ssh controller nova-manage service list | ");
375     while( my $line = <SSH> ){
376         if ( $line =~ /\s*nova-compute\s+(\S+)\s+nova\s+(\S+)\s+(\S+)\s+\d/ ){
377             $computestatus{$1}{"status"} = $2;
378             $computestatus{$1}{"face"} = $3;
379             my $compute_name = $1;
380             $compute_name =~ /[a-z]+0*(\d+)$/;
381             my $compute_number = $1;
382             $computestatus{$compute_name}{"compute_number"}=$compute_number;
383             $maxrunningcompute = $compute_number if ( $compute_number >
384 $maxrunningcompute and  $computestatus{$compute_name}{"face"} eq ":-)" and
385 $computestatus{$compute_name}{"status"} eq "enabled");
```

```
386         }
387     }
388
389 }
390
391 sub runMigrations {
392
393     foreach my $vm ( keys %vms){
394
395         if ( $vms{$vm} ne $vms_old{$vm} ){
396             print "Migrating $nametable{$vm}: $vms_old{$vm} -> $vms{$vm}\n";
397             # ssh -t controller "source creds; nova live-migration
398 ad1e2027-5eca-492a-ad3e-872fc086e7dd compute06
399             if ( $nametable{$vm} =~ /fiobench/ ){
400                 print "All systems GO!\n";
401                 # system("ssh -t controller 'source creds; nova live-migration $vm
402 $vms{$vm}'");
403                 # while( getVMLocation($vm) ne $vms{$vm}){
404                 #     verbose("sleeping...\n");
405                 #     sleep $MIGRATIONPAUSE;
406                 # }
407             }
408         }
409
410     }
411
412 }
```

9.2 Policy 2

Policy2.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables

my $VERBOSE = 0;
my $DEBUG = 0;
my $MIGRATIONPAUSE = 3;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname
my %status;
my %computestatus;
my %vm_count;
my %compute_receive;
my %vms;
my %vms_old;
my %nametable;
my %commands;
my %important_vcpu;
my %saved_nodes;
my $maxrunningcompute = 0;
my $runningcompute = 0;
my $buffer = 1;
my @running_compute;
my @stopped_compute;
my @error_compute;
my @not_important;
my $targetnode;
my $vacanter;
my $vcpu_subtract;
my $vcpu_nr;
my $livemigration;
# my $important_vcpu;
# my $not_important_vcpu;

open(MYFILE, 'notimportant.txt');


##### Importance levels ########

foreach my $line (<MYFILE>){

    chomp($line);

     push(@not_important, $line)

}

close(MYFILE);


my $high_cpu =
"5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40";
my $low_cpu =
"41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64";

# command line options
```

```
66
67  my $opt_string = "vdh";
68  getopts("$opt_string", \my %opt) or usage() and exit(1);
69
70  $VERBOSE = 1 if $opt{'v'};
71  $DEBUG = 1 if $opt{'d'};
72
73  if($opt{'h'}){
74          usage();
75          exit 0;
76  }
77
78
79  ##################################################
80  # Main part
81
82  getstatus();
83  getservicestatus();
84  getcomputestatus();
85
86
87  %vms_old = %vms;
88  # printstatus();
89  # printcompute();
90
91  # run algorithm which presents new status
92  # my $crosspoint = agressiveRelocation();
93
94  # printstatus();
95
96
97
98  # printcompute();
99
100 policy2();
101 printstatus();
102
103 runMigrations();
104 scaleservers();
105
106 # printcommands();
107
108  countmigrations();
109  printimportant();
110 ######################################## CPU allocation ########################
111
112
113
114 ##################################################
115 # Subroutines
116
117 sub scaleservers {
118
119 #     my $crosspoint = $_[0];
120
121 #    print "Crosspoint was at $crosspoint\n";
122     # if crosspoint + buffer < maxrunning
123     print "Running compute nodes: $runningcompute\n";
124     print "Number of saved nodes: " . (scalar keys %saved_nodes) . "\n";
125     foreach my $compute ( keys %saved_nodes ){
126         print "This compute is saved: $compute\n";
127     }
128     print "Number of saved nodes: " . (scalar keys %compute_receive) . "\n";
129
130     if ( $runningcompute > ( scalar keys %saved_nodes) ){
131         my @stop_candidates;
132         foreach my $compute ( keys %computestatus ){
133             next unless $compute;
```

```
134             if ( not $saved_nodes{$compute} ){
135                 print "potential target for shutdown: '$compute'\n";
136                 system("ssh $compute service nova-compute stop");
137                 push(@stop_candidates,$compute);
138             }
139
140         }
141
142         for ( my $i = 0; $i < $buffer; $i++ ){
143             print "Sparing from shutown: " . pop(@stop_candidates) . "\n";
144         }
145
146         foreach my $compute (@stop_candidates){
147             print "target for shutdown: $compute\n";
148             system("ssh $compute service nova-compute stop");
149         }
150     }
151
152  return;
153     # if( ($crosspoint + $buffer) < $maxrunningcompute){
154     #         print "We are using too many compute nodes, scaling down\n";
155     #         for ( my $i = ( $crosspoint + $buffer + 1 ); $i <= $maxrunningcompute;
156 $i++ ){
157     #             my $number = $i;
158     #             $number = "0" . $number if $number < 10;
159     #             my $computename = "compute$number";
160     #             print "shutting down $computename\n";
161     #             system("ssh $computename service nova-compute stop");
162     #         }
163     # # open(SSH,"ssh controller nova-manage service list | ");
164
165     # }elsif(($crosspoint + $buffer) > $maxrunningcompute){
166     #         print "we are under capacity, scaling up\n";
167     #         my $number = $maxrunningcompute + 1;
168     #         $number = "0" . $number if $number < 10;
169     #         my $computename = "compute$number";
170     #         print "starting $computename\n";
171     #         system("ssh $computename service nova-compute start");
172
173     # }
174 }
175
176
177 sub getVMLocation {
178     my $vm = $_[0];
179     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
180 $password)
181       || die "Could not connect to database: $DBI::errstr";
182     my $sth = $dbh->prepare("select * from instances where uuid ='$vm';")
183       || die "$DBI::errstr";
184     $sth->execute();
185     my $results = $sth->fetchrow_hashref;
186     $sth->finish;
187     $dbh->disconnect;
188
189     return $results->{"host"};
190
191 }
192
193 sub getstatus {
194
195     # Get the rows from database
196     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
197 $password)
198       || die "Could not connect to database: $DBI::errstr";
199     my $sth = $dbh->prepare('select * from instances where vm_state="active";')
200       || die "$DBI::errstr";
201     $sth->execute();
```

```
202
203     # Print number of rows found
204     if ($sth->rows < 0) {
205         print "Sorry, no vm found failure in getstatus subrutine.\n";
206     } else {
207         # printf ">> Found something\n", $sth->rows;
208         # Loop if results found
209         while (my $results = $sth->fetchrow_hashref) {
210
211           my $vm_name = $results->{uuid}; # get the hostname
212           my $computenode = $results->{host}; # get the computenode
213           my $vm_vcpu = $results->{vcpus}; # VM number of cpu
214           my $vm_memory = $results->{memory_mb}; # VM number of memory
215           my $vm_project_id = $results->{project_id}; # VM number of memory
216         #    my $vm_uuid = $results->{uuid}; # VM uuid
217
218
219           #   printf  $vm_name . " at ". $computenode . " with VCPU: " . $vm_vcpu .
220     " and memory: " . $vm_memory . "\n";
221
222           $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
223           $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
224           $status{$computenode}{$vm_name}{"project_id"}=$vm_project_id;
225         #   $status{$computenode}{$vm_name}{"uuid"}=$vm_uuid;
226
227           $vms{$results->{uuid}} = $computenode;
228           $nametable{$results->{uuid}} = $results->{"hostname"};
229
230
231
232           if ($vm_name ~~ @not_important) {
233               $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
234     #              print "NTANT\n"; #  " . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
235               $important_vcpu{$computenode}{"not_important_vcpu"} =
236     ($important_vcpu{$computenode}{"not_important_vcpu"} + $vcpu_nr);
237
238
239           }else{
240
241               $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
242
243     #                print "IMPORTANT\n"; # . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
244                $important_vcpu{$computenode}{"important_vcpu"} =
245     ($important_vcpu{$computenode}{"important_vcpu"} + $vcpu_nr);
246
247
248           }
249
250
251       }
252
253     # Disconnect
254         $sth->finish;
255         $dbh->disconnect;
256     }
257     }
258
259     sub getcomputestatus {
260
261     # Get the rows from database
262     my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username,
263     $password)
264       || die "Could not connect to database: $DBI::errstr";
265     my $sth2 = $dbh2->prepare('select * from compute_nodes')
266       || die "$DBI::errstr";
267     $sth2->execute();
268
269     # Print number of rows found
```

```
    if ($sth2->rows < 0) {
        print "Sorry, no vm found failure in getstatus subrutine.\n";
    } else {
        # printf ">> Found something\n", $sth->rows;
        # Loop if results found
        while (my $results2 = $sth2->fetchrow_hashref) {


            my $compute_name = $results2->{hypervisor_hostname}; # get the hostname
            my $compute_vcpu = $results2->{vcpus}; # VM number of cpu
            my $compute_memory = $results2->{memory_mb}; # VM number of memory
            my $compute_vcpu_used = $results2->{vcpus_used}; # VM number of vcpu used
            my $compute_memory_used = $results2->{memory_mb_used}; # VM number of vcpu
used
            my $compute_running_vms = $results2->{running_vms}; # VM number of vcpu
used
            my $compute_free_memory = $compute_memory - $compute_memory_used;
            my $compute_free_vcpu = 64 - $compute_vcpu_used;


            # print  $compute_name . " running VM: " . $compute_running_vms . " number
of used vcpu : " . $compute_vcpu_used . "free VCPU: " . $compute_free_vcpu  .
"\n";
            if ( $computestatus{$compute_name} ){
                $computestatus{$compute_name}{"running vms"}=$compute_running_vms;
                $computestatus{$compute_name}{"compute_free_vcpu"}=$compute_free_vcpu;
                $computestatus{$compute_name}{"compute_free_memory"}=$compute_free_memory;


                $vm_count{$compute_name}=$compute_running_vms;
                $compute_receive{$compute_name}=0;
                $commands{$compute_name};
            }

        }
    }

# Disconnect
    $sth2->finish;
    $dbh2->disconnect;
}


sub printcompute {

    foreach my $computenode (sort(keys %computestatus)) {
        print "$computenode: \n";
        foreach my $item (keys %{$computestatus{$computenode}}) {
            print "\t$item: ".$computestatus{$computenode}{$item}."\n";

        }
    }
}

sub printimportant {

    foreach my $computenode (sort(keys %important_vcpu)) {
        print "$computenode: \n";
        foreach my $item (keys %{$important_vcpu{$computenode}}) {
            print "\t$item: ".$important_vcpu{$computenode}{$item}."\n";

        }
    }
}



sub printstatus {
```

```
338     foreach my $computenode (keys %status) {
339        print "$computenode: \n";
340        foreach my $vm (keys %{$status{$computenode}}) {
341           print "\t vm-name:$vm\n";
342           foreach my $noe (keys %{$status{$computenode}{$vm}}) {
343              print "\t$noe: " . $status{$computenode}{$vm}->{$noe} . "\n";
344
345
346           }
347        }
348     }
349  }
350
351
352
353
354
355  sub getLoggingTime {
356
357     my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
358     my $nice_timestamp = sprintf ( "%04d.%02d.%02d:%02d:%02d:%02d",
359                                    $year+1900,$mon+1,$mday,$hour,$min,$sec);
360     return $nice_timestamp;
361  }
362
363
364  sub usage {
365
366        print "Usage:\n";
367        print "-h for help\n";
368        print "-v for verbose(more output)\n";
369        print "-d for debug(even more output)\n";
370
371  }
372
373  sub verbose {
374        print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
375  }
376
377  sub debug {
378        print "DEBUG: " . $_[0] if ($DEBUG);
379  }
380
381  sub getservicestatus {
382
383     # nova-compute     compute09                          nova
384  enabled    :-)   2014-03-21 19:51:32
385     open(SSH,"ssh controller nova-manage service list | ");
386     while( my $line = <SSH> ){
387        if ( $line =~ /\s*nova-compute\s+(\S+)\s+nova\s+(\S+)\s+(\S+)\s+\d/ ){
388           my $face = $3;
389           my $status = $2;
390           my $compute_name = $1;
391           if ( $status eq "enabled" and $face eq ":-)" ){
392              $computestatus{$1}{"status"} = $status;
393              $computestatus{$1}{"face"} = $face;
394
395              $compute_name =~ /[a-z]+0*(\d+)$/;
396
397  #              $computestatus{$compute_name}{"compute_number"} = $compute_number;
398  #              $maxrunningcompute = $compute_number if ( $compute_number >
399  $maxrunningcompute and  $computestatus{$compute_name}{"face"} eq ":-)" and
400  $computestatus{$compute_name}{"status"} eq "enabled");
401              $runningcompute++ if ( $computestatus{$compute_name}{"face"} eq ":-)" and
402  $computestatus{$compute_name}{"status"} eq "enabled");
403           } elsif( $face eq "XXX" and $status eq "enabled") {
404  #              push(@
405              push(@stopped_compute, $compute_name);
```

```
406             }
407         }
408     }
409
410 }
411
412 sub runMigrations {
413
414     foreach my $vm ( keys %vms){
415
416         if ( $vms{$vm} ne $vms_old{$vm} ){
417             print "Migrating $nametable{$vm}: $vms_old{$vm} -> $vms{$vm}\n";
418             # ssh -t controller "source creds; nova live-migration
419 ad1e2027-5eca-492a-ad3e-872fc086e7dd compute06
420 #           if ( $nametable{$vm} =~ /fiobench/ ){
421             print "All systems GO!\n";
422             system("ssh -t controller 'source creds; nova live-migration $vm
423 $vms{$vm}'");
424             while( getVMLocation($vm) ne $vms{$vm}){
425                 verbose("sleeping...\n");
426                 sleep $MIGRATIONPAUSE;
427             }
428         }
429     }
430
431 }
432
433 # }
434
435
436 sub policy2 {
437
438     print "policy2 algorithm\n";
439     print "Sorting computes after number of vms". "\n";
440
441     foreach my $compute (sort { $vm_count{$b} <=> $vm_count{$a} } keys %vm_count)
442 {
443         printf "%-8s %s\n", $compute, $vm_count{$compute};
444         if (($computestatus{$compute}{"status"} =~ /enable/) and
445 ($computestatus{$compute}{"face"} =~ /\:\-\)/ )){
446             # print  $compute . " VMs:" . $vm_count{$compute} . " status: " .
447 $computestatus{$compute}{"status"} . " og " . $computestatus{$compute}{"face"}.
448 "\n";
449             # print "$compute is sorted, running and placed in running_compute array
450 \n";
451             push(@running_compute, $compute)
452         }elsif (($computestatus{$compute}{"status"} =~ /enable/) and
453 ($computestatus{$compute}{"face"} !=~ /\:\-\)/ )){
454             # print "$compute NOT running and placed in stopped_compute array \n";
455             push(@stopped_compute, $compute)
456         }elsif (($computestatus{$compute}{"status"} =~ /enable/) or
457 ($computestatus{$compute}{"face"} !=~ /\:\-\)/ )){
458             # print "$compute has an ERROR.. Placed in error_compute array \n";
459             push(@error_compute, $compute)
460         }
461     }
462
463
464     my $t = 0;
465     my $v = $#running_compute;
466     my $runde = 1;
467     my $last_compute;
468
469     foreach my $computenode (sort { $vm_count{$b} <=> $vm_count{$a} } keys
470 %vm_count) {
471         # printf "%-8s %s\n", $compute, $vm_count{$compute};
472
473         $targetnode = @running_compute[$t];
```

```
474         $vacanter = @running_compute[$v];
475         $last_compute = $computenode;
476
477
478         # Quit if vacanter and target is the same
479         last if ($compute_receive{$vacanter} > 0);
480
481         print "From $vacanter to $targetnode:\n";
482
483         # Trying to move all vms from vacanter
484         foreach my $vm ( keys %{$status{$vacanter}}){
485
486            # $targetnode = @running_compute[$t];
487            $vacanter = @running_compute[$v];
488
489            foreach my $targetnode (sort { $vm_count{$b} <=> $vm_count{$a} } keys
490    %vm_count) {
491
492
493
494               print "\tTesting for vm " . $nametable{$vm} . " from " . $vacanter . " to
495    " . $targetnode  . "\n";
496
497               my $cpu_ok = 0;
498               my $mem_ok = 0;
499
500               if ($vm ~~ @not_important) {
501                  $vcpu_nr = ($status{$vacanter}{$vm}{"vcpu"});
502                 # aaa
503                  $vcpu_subtract = ($vcpu_nr/16);
504                  # $vcpu_subtract = ($vcpu_nr);
505                  # print "Not important VM detected: " . $vm . "VCPU: " . ($vcpu_nr+1)
506    . " \n";
507
508               }else{
509                  $vcpu_nr = ($status{$vacanter}{$vm}{"vcpu"});
510                  $vcpu_subtract = ($vcpu_nr);
511                  #  print "IMPORTANT:  " . $vm . " Vcpu: " . ($vcpu_nr+1) .
512    "\n";
513
514               }
515
516
517
518               # target has enough vcpu
519               #    if ( $computestatus{$targetnode}{"compute_free_vcpu"} >=
520    $status{$vacanter}{$vm}{"vcpu"} ){
521               if ( $computestatus{$targetnode}{"compute_free_vcpu"} >= $vcpu_subtract ){
522                  $cpu_ok = 1;
523               }
524
525               # target has enough memory
526               if ( $computestatus{$targetnode}{"compute_free_memory"} >=
527    $status{$vacanter}{$vm}{"memory"} ){
528                  $mem_ok = 1;
529               }
530
531
532
533
534
535               if ( $cpu_ok and $mem_ok ){
536                  print "\tVM will fit in, scheduling migration to $targetnode\n";
537                  $saved_nodes{$targetnode} = 1;
538                  # Updating number of moved VMs
539                  $compute_receive{$targetnode} = ($compute_receive{$targetnode} +1);
540
541                  # update %vms
```

```
542                 $vms{$vm} = $targetnode;
543
544                 # reduce compute_free_mem on target
545                 $computestatus{$targetnode}{"compute_free_memory"} =
546 $computestatus{$targetnode}{"compute_free_memory"} -
547 $status{$vacanter}{$vm}{"memory"};
548
549                 # reduce important compute_free_vcpu on target
550                 # $computestatus{$targetnode}{"compute_free_vcpu"} =
551 $computestatus{$targetnode}{"compute_free_vcpu"} -
552 $status{$vacanter}{$vm}{"vcpu"};
553
554                 # reducing a NOT important vm from compute_free_vcpu
555                 # $computestatus{$targetnode}{"compute_free_vcpu"} =
556 $computestatus{$targetnode}{"compute_free_vcpu"} -
557 ($status{$vacanter}{$vm}{"vcpu"} / 16);
558                 $computestatus{$targetnode}{"compute_free_vcpu"} =
559 ($computestatus{$targetnode}{"compute_free_vcpu"} - $vcpu_subtract);
560
561                 # add vm to $status{$target}
562                 $status{$targetnode}{$vm} = $status{$vacanter}{$vm};
563
564                 # remove vm from $status{$computenode}
565                 delete $status{$vacanter}{$vm};
566
567
568
569                 if ($vm ~~ @not_important) {
570
571                    my $i = 0;
572                    while ($i <= $vcpu_nr){
573                       # system("ssh $node virsh vcpupin $line $i $low_cpu");
574                       # print "\tssh $targetnode virsh vcpupin $vm $i $low_cpu \n";
575                       $commands{$targetnode} .= "virsh vcpupin $vm $i low_cpu; ";
576                       $i++;
577                    }
578
579                 }else{
580                    my $i = 0;
581                    while ($i <= $vcpu_nr){
582                       # system("ssh $node virsh vcpupin $line $i $low_cpu");
583                       # print "\tssh $targetnode virsh vcpupin $vm $i high_cpu \n";
584                       $commands{$targetnode} .= "virsh vcpupin $vm $i high_cpu; ";
585                       $i++;
586                       #  print "command er naa= " . $commands{$targetnode};
587                    }
588                 }
589
590                 last;
591              }else{
592
593                 print "Target is full, changing to next targetnode\n";
594                 # increasing target to the next compute node in array sorted after
595 running vms
596                 $t=$t+1;
597
598
599              }
600           }
601        }
602
603 # decreasing vacanter to the next compute node in sorted array after running vms
604        $v--;
605
606        print "$targetnode receive =  " . $compute_receive{$targetnode} . " status:
607 free_cpu: " . $computestatus{$targetnode}{"compute_free_vcpu"} . "\n";
608
609     }
```

```
610     print "saving last compute: $last_compute\n";
611 #    $saved_nodes{$last_compute} = 1;
612
613 }
614
615 sub printcommands {
616     foreach my $computenode (keys %commands) {
617         print "\n\nssh $computenode " . $commands{$computenode};
618         }
619 }
620
621 sub countmigrations {
622
623     foreach my $computenode (keys %compute_receive) {
624         $livemigration = ($livemigration + $compute_receive{$computenode});
625     }
626
627     print "\n\nThe number of live migration is: " . $livemigration . "\n";
628 }
```

9.3 Policy 3

Policy3.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables
my $old_spot;
my $VERBOSE = 0;
my $DEBUG = 0;
my $MIGRATIONPAUSE = 5;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname my %status;
my %computestatus;
my %vm_count;
my %compute_receive;
my %vms;
my %vms_old;
my %nametable;
my %commands;
my %important_vcpu;
my %saved_nodes;
my $maxrunningcompute = 0;
my $runningcompute = 0;
my $buffer = 1;
my @running_compute;
my @stopped_compute;
my @error_compute;
my @not_important;
my $targetnode;
my $vacanter;
my $vcpu_subtract;
my $vcpu_nr;
my $livemigration;
my %order;
my $mem;
my $live;
# my $important_vcpu;
# my $not_important_vcpu;

# number of items to pack
my $numberofvms;
# dimensions of bin
my $w = 8;
my $h = 8;
my $d = 6;

open(MYFILE, 'notimportant.txt');
open(FILE, '<file4.txt');
open(HVOR, '>location.txt');
##### Importance levels ########

foreach my $line (<MYFILE>){

    chomp($line);

     push(@not_important, $line)

}

close(MYFILE);
```

```
66
67
68  my $high_cpu =
69  "5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
70  ,34,35,36,37,38,39,40";
71  my $low_cpu =
72  "41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64";
73
74  # command line options
75
76  my $opt_string = "vdh";
77  getopts("$opt_string", \my %opt) or usage() and exit(1);
78
79  $VERBOSE = 1 if $opt{'v'};
80  $DEBUG = 1 if $opt{'d'};
81
82  if($opt{'h'}){
83        usage();
84        exit 0;
85  }
86
87
88  ##################################################
89  # Main part
90
91  getstatus();
92  getservicestatus();
93  getcomputestatus();
94
95
96  %vms_old = %vms;
97  # printstatus();
98  # printcompute();
99
100 # run algorithm which presents new status
101 # my $crosspoint = agressiveRelocation();
102 my $implevel;
103 my $k = 1;
104
105 # my $dbh3 = DBI->connect("DBI:mysql:$database;host=$server", $username,
106 $password)
107 #     || die "Could not connect to database: $DBI::errstr";
108 # my $sth3 = $dbh3->prepare('select * from instances where vm_state="active";')
109 #     || die "$DBI::errstr";
110 # $sth3->execute();
111
112  #   $sth3->finish;
113  #   $dbh3->disconnect;
114
115 print FILE "202 $w $h $d\n";
116
117 foreach my $computenode (keys %status) {
118     foreach my $vm (keys %{$status{$computenode}}) {
119         if ($vm ~~ @not_important) {
120             $implevel = "1";
121             }else{
122             $implevel = "2";
123
124         }
125
126         # print FILE $status{$computenode}{$vm}{'vcpu'} . " " .
127 ($status{$computenode}{$vm}{'memory'}/1000) . " " . $implevel . "\n";
128
129
130         $mem = $status{$computenode}{$vm}{'memory'};
131         if ($mem =~ /512/){
132             $mem = 1;
133         }elsif($mem =~ /2048/){
```

```
134             $mem=2;
135         }elsif($mem =~ /4096/){
136                 $mem=4;
137         }elsif($mem =~ /8192/){
138                 $mem=8;
139         }elsif($mem =~ /16384/){
140                 $mem=16;
141         }
142
143
144         print FILE $status{$computenode}{$vm}{'vcpu'} . " " . $mem . " " . $implevel .
145 "\n";
146         $order{$vm}{"plass"}=$k;
147         $k++;
148     }
149 }
150
151 foreach my $vm (keys %order) {
152     foreach my $nr (keys %{$order{$vm}}) {
153         print "\t$vm: " . "plass=". $order{$vm}{'plass'} . "\n";
154                 print HVOR "$vm: " . "plass=". $order{$vm}{'plass'} . "\n";
155     }
156 }
157
158
159
160 close (FILE);
161 close (HVOR);
162
163 my @response = `./3dbpp file4.txt 0 0 0 0 | grep Bin | cut -f 2 -d ":" | cut -f 4
164 -d " "`;
165 print @response;
166
167 foreach my $vm (keys %order) {
168     foreach my $nr (keys %{$order{$vm}}) {
169         # print "\t$vm: " . "plass=". $order{$vm}{'plass'} . "\n";
170         my $plass = $order{$vm}{'plass'};
171         my $res =  @response[$plass];
172         # aa
173         my $nodecode = "compute0$res";
174         chomp($nodecode);
175
176         if( getVMLocation($vm) ne $nodecode){
177                 print  "$vm:  from $old_spot to $nodecode\n";
178             $live++;
179         }
180
181
182     }
183 }
184
185 print "\nNumber of live migrations required for policy3 = " . $live . "\n";
186
187 ##################################################
188 # Subroutines
189
190 sub scaleservers {
191
192 #     my $crosspoint = $_[0];
193
194 #    print "Crosspoint was at $crosspoint\n";
195     # if crosspoint + buffer < maxrunning
196     print "Running compute nodes: $runningcompute\n";
197     print "Number of saved nodes: " . (scalar keys %saved_nodes) . "\n";
198     foreach my $compute ( keys %saved_nodes ){
199         print "This compute is saved: $compute\n";
200     }
201     print "Number of saved nodes: " . (scalar keys %compute_receive) . "\n";
```

```
202
203     if ( $runningcompute > ( scalar keys %saved_nodes) ){
204         my @stop_candidates;
205         foreach my $compute ( keys %computestatus ){
206             next unless $compute;
207             if ( not $saved_nodes{$compute} ){
208                 print "potential target for shutdown: '$compute'\n";
209 #                system("ssh $compute service nova-compute stop");
210                 push(@stop_candidates,$compute);
211             }
212
213         }
214
215         for ( my $i = 0; $i < $buffer; $i++ ){
216             print "Sparing from shutown: " . pop(@stop_candidates) . "\n";
217         }
218
219         foreach my $compute (@stop_candidates){
220             print "target for shutdown: $compute\n";
221             system("ssh $compute service nova-compute stop");
222         }
223     }
224
225  return;
226     # if( ($crosspoint + $buffer) < $maxrunningcompute){
227     #         print "We are using too many compute nodes, scaling down\n";
228     #         for ( my $i = ( $crosspoint + $buffer + 1 ); $i <= $maxrunningcompute;
229 $i++ ){
230     #             my $number = $i;
231     #             $number = "0" . $number if $number < 10;
232     #             my $computename = "compute$number";
233     #             print "shutting down $computename\n";
234     #             system("ssh $computename service nova-compute stop");
235     #         }
236     # # open(SSH,"ssh controller nova-manage service list | ");
237
238     # }elsif(($crosspoint + $buffer) > $maxrunningcompute){
239     #         print "we are under capacity, scaling up\n";
240     #         my $number = $maxrunningcompute + 1;
241     #         $number = "0" . $number if $number < 10;
242     #         my $computename = "compute$number";
243     #         print "starting $computename\n";
244     #         system("ssh $computename service nova-compute start");
245
246     # }
247 }
248
249
250 sub getVMLocation {
251     my $vm = $_[0];
252     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
253 $password)
254       || die "Could not connect to database: $DBI::errstr";
255     my $sth = $dbh->prepare("select * from instances where uuid ='$vm';")
256       || die "$DBI::errstr";
257     $sth->execute();
258     my $results = $sth->fetchrow_hashref;
259     $sth->finish;
260     $dbh->disconnect;
261     $old_spot = $results->{"host"};
262     return $results->{"host"};
263
264 }
265
266 sub getstatus {
267
268     # Get the rows from database
269     my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
```

```
$password)
    || die "Could not connect to database: $DBI::errstr";
  my $sth = $dbh->prepare('select * from instances where vm_state="active";')
    || die "$DBI::errstr";
  $sth->execute();
  $numberofvms = $sth->rows;

  # Print number of rows found
  if ($sth->rows < 0) {
    print "Sorry, no vm found failure in getstatus subrutine.\n";
  } else {
    # printf ">> Found something\n", $sth->rows;
    # Loop if results found
    while (my $results = $sth->fetchrow_hashref) {

      my $vm_name = $results->{uuid}; # get the hostname
      my $computenode = $results->{host}; # get the computenode
      my $vm_vcpu = $results->{vcpus}; # VM number of cpu
      my $vm_memory = $results->{memory_mb}; # VM number of memory
      my $vm_project_id = $results->{project_id}; # VM number of memory
    #    my $vm_uuid = $results->{uuid}; # VM uuid


       #   printf  $vm_name . " at ". $computenode . " with VCPU: " . $vm_vcpu .
" and memory: " . $vm_memory . "\n";

       $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
       $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
       $status{$computenode}{$vm_name}{"project_id"}=$vm_project_id;
     #   $status{$computenode}{$vm_name}{"uuid"}=$vm_uuid;

       $vms{$results->{uuid}} = $computenode;
       $nametable{$results->{uuid}} = $results->{"hostname"};



       if ($vm_name ~~ @not_important) {
          $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});

          $important_vcpu{$computenode}{"not_important_vcpu"} =
($important_vcpu{$computenode}{"not_important_vcpu"} + $vcpu_nr);


       }else{

          $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});

          #  print "IMPORTANT:  " . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
           $important_vcpu{$computenode}{"important_vcpu"} =
($important_vcpu{$computenode}{"important_vcpu"} + $vcpu_nr);


       }


    }

# Disconnect
    $sth->finish;
    $dbh->disconnect;
  }
}

sub getcomputestatus {

  # Get the rows from database
  my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username,
$password)
```

```
338       || die "Could not connect to database: $DBI::errstr";
339     my $sth2 = $dbh2->prepare('select * from compute_nodes')
340       || die "$DBI::errstr";
341     $sth2->execute();
342
343   # Print number of rows found
344     if ($sth2->rows < 0) {
345         print "Sorry, no vm found failure in getstatus subrutine.\n";
346     } else {
347         # printf ">> Found something\n", $sth->rows;
348         # Loop if results found
349         while (my $results2 = $sth2->fetchrow_hashref) {
350
351
352             my $compute_name = $results2->{hypervisor_hostname}; # get the hostname
353             my $compute_vcpu = $results2->{vcpus}; # VM number of cpu
354             my $compute_memory = $results2->{memory_mb}; # VM number of memory
355             my $compute_vcpu_used = $results2->{vcpus_used}; # VM number of vcpu used
356             my $compute_memory_used = $results2->{memory_mb_used}; # VM number of vcpu
357   used
358             my $compute_running_vms = $results2->{running_vms}; # VM number of vcpu
359   used
360             my $compute_free_memory = $compute_memory - $compute_memory_used;
361             my $compute_free_vcpu = 64 - $compute_vcpu_used;
362
363
364             # print  $compute_name . " running VM: " . $compute_running_vms . " number
365   of used vcpu : " . $compute_vcpu_used . "free VCPU: " . $compute_free_vcpu  .
366   "\n";
367             if ( $computestatus{$compute_name} ){
368                 $computestatus{$compute_name}{"running vms"}=$compute_running_vms;
369                 $computestatus{$compute_name}{"compute_free_vcpu"}=$compute_free_vcpu;
370                 $computestatus{$compute_name}{"compute_free_memory"}=$compute_free_memory;
371
372
373                 $vm_count{$compute_name}=$compute_running_vms;
374                 $compute_receive{$compute_name}=0;
375                 $commands{$compute_name};
376             }
377
378         }
379     }
380
381   # Disconnect
382     $sth2->finish;
383     $dbh2->disconnect;
384   }
385
386
387   sub printcompute {
388
389     foreach my $computenode (sort(keys %computestatus)) {
390         print "$computenode: \n";
391         foreach my $item (keys %{$computestatus{$computenode}}) {
392             print "\t$item: ".$computestatus{$computenode}{$item}."\n";
393
394         }
395     }
396   }
397
398   sub printimportant {
399
400     foreach my $computenode (sort(keys %important_vcpu)) {
401         print "$computenode: \n";
402         foreach my $item (keys %{$important_vcpu{$computenode}}) {
403             print "\t$item: ".$important_vcpu{$computenode}{$item}."\n";
404
405         }
```

```
406     }
407 }
408
409
410
411 sub printstatus {
412     foreach my $computenode (keys %status) {
413         print "$computenode: \n";
414         foreach my $vm (keys %{$status{$computenode}}) {
415             print "\t vm-name:$vm\n";
416             foreach my $noe (keys %{$status{$computenode}{$vm}}) {
417                 print "\t$noe: " . $status{$computenode}{$vm}->{$noe} . "\n";
418
419
420             }
421         }
422     }
423 }
424
425
426
427
428
429
430 sub getLoggingTime {
431
432     my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
433     my $nice_timestamp = sprintf ( "%04d.%02d.%02d:%02d:%02d:%02d",
434                                    $year+1900,$mon+1,$mday,$hour,$min,$sec);
435     return $nice_timestamp;
436 }
437
438
439 sub usage {
440
441         print "Usage:\n";
442         print "-h for help\n";
443         print "-v for verbose(more output)\n";
444         print "-d for debug(even more output)\n";
445
446 }
447
448 sub verbose {
449         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
450 }
451
452 sub debug {
453         print "DEBUG: " . $_[0] if ($DEBUG);
454 }
455
456 sub getservicestatus {
457
458     # nova-compute     compute09                       nova
459 enabled    :-)   2014-03-21 19:51:32
460     open(SSH,"ssh controller nova-manage service list | ");
461     while( my $line = <SSH> ){
462         if ( $line =~ /\s*nova-compute\s+(\S+)\s+nova\s+(\S+)\s+(\S+)\s+\d/ ){
463             my $face = $3;
464             my $status = $2;
465             my $compute_name = $1;
466             if ( $status eq "enabled" and $face eq ":-)" ){
467                 $computestatus{$1}{"status"} = $status;
468                 $computestatus{$1}{"face"} = $face;
469
470                 $compute_name =~ /[a-z]+0*(\d+)$/;
471
472 #                 $computestatus{$compute_name}{"compute_number"} = $compute_number;
473 #                 $maxrunningcompute = $compute_number if ( $compute_number >
```

```
474 $maxrunningcompute and  $computestatus{$compute_name}{"face"} eq ":-)" and
475 $computestatus{$compute_name}{"status"} eq "enabled");
476             $runningcompute++ if ( $computestatus{$compute_name}{"face"} eq ":-)" and
477 $computestatus{$compute_name}{"status"} eq "enabled");
478         } elsif( $face eq "XXX" and $status eq "enabled") {
479 #             push(@
480             push(@stopped_compute, $compute_name);
481         }
482       }
483   }
484
485 }
486
487 sub runMigrations {
488
489   foreach my $vm ( keys %vms){
490
491       if ( $vms{$vm} ne $vms_old{$vm} ){
492         print "Migrating $nametable{$vm}: $vms_old{$vm} -> $vms{$vm}\n";
493         # ssh -t controller "source creds; nova live-migration
494 ad1e2027-5eca-492a-ad3e-872fc086e7dd compute06
495 #          if ( $nametable{$vm} =~ /fiobench/ ){
496        #   print "All systems GO!\n";
497         system("ssh -t controller 'source creds; nova live-migration $vm
498 $vms{$vm}'");
499         while( getVMLocation($vm) ne $vms{$vm}){
500           verbose("sleeping...\n");
501           sleep $MIGRATIONPAUSE;
502         }
503 #         }
504       }
505
506   }
507
508 }
509
510 sub printcommands {
511   foreach my $computenode (keys %commands) {
512     print "\n\nssh $computenode " . $commands{$computenode};
513     }
514 }
515
516 sub countmigrations {
517
518   foreach my $computenode (keys %compute_receive) {
519     $livemigration = ($livemigration + $compute_receive{$computenode});
520   }
521
522   print "\n\nThe number of live migration is: " . $livemigration . "\n";
523 }
```

9.4 CPU affinity script

CPU affinity script

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables

my $lq;
my $hq;
my $only_q;
my $only_computenode;
my $only_vm;
my $only_command;
my $send_command;
my $stopfile = '/home/thomas/powerconsumption/policy_disable';
my $VERBOSE = 0;
my $DEBUG = 0;
my $simulate = 0;
my $check = 0;
my $uuid;
my $range;
my $MIGRATIONPAUSE = 3;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname
my %status;
my %computestatus;
my %vm_count;
my %compute_receive;
my %vms;
my %vms_old;
my %nametable;
my %commands;
my %important_vcpu;
my %saved_nodes;
my $maxrunningcompute = 0;
my $runningcompute = 0;
my $buffer = 1;
my @running_compute;
my @stopped_compute;
my @error_compute;
my @not_important;
my $targetnode;
my $vacanter;
my $vcpu_subtract;
my $vcpu_nr;
my $livemigration;
# my $important_vcpu;
# my $not_important_vcpu;



open(MYFILE, 'notimportant.txt');


##### Importance levels ########

foreach my $line (<MYFILE>){

    chomp($line);

```

```
66      push(@not_important, $line)
67
68  }
69
70  close(MYFILE);
71
72  # command line options
73
74  my $opt_string = "vdhau:r:H:L:c";
75  getopts("$opt_string", \my %opt) or usage() and exit(1);
76
77  $VERBOSE = 1 if $opt{'v'};
78  $DEBUG = 1 if $opt{'d'};
79  $simulate = 1 if $opt{'s'};
80  $check = 1 if $opt{'c'};
81
82  # High range
83  my $Hrange = $opt{'H'};
84  if($opt{'H'}){
85      $Hrange =~  (/(\d+)-(\d+)/);
86      my $Hrange_low = $1;
87      my $Hrange_high = $2;
88   #  $Hrange_low = ($Hrange_low - 1);
89   #  $Hrange_high = ($Hrange_high - 1);
90
91
92      print "range = " . $Hrange_low . "\n";
93      print "range = " . $Hrange_high . "\n";
94      for($Hrange_low; $Hrange_low <= $Hrange_high; $Hrange_low++){
95          $hq .= "$Hrange_low,";
96      }
97  }
98
99
100 # Low range
101 my $Lrange = $opt{'L'};
102 if($opt{'L'}){
103     $Lrange =~  (/(\d+)-(\d+)/);
104     my $Lrange_low = $1;
105     my $Lrange_high = $2;
106 #    $Lrange_low = ($Lrange_low - 1);
107  #   $Lrange_high = ($Lrange_high -1);
108     print "low q range = " . $Lrange_low . "\n";
109     print "Low q range = " . $Lrange_high . "\n";
110     for($Lrange_low; $Lrange_low <= $Lrange_high; $Lrange_low++){
111       $lq .= "$Lrange_low,";
112     }
113 }
114
115
116
117 chop($lq);
118 chop($hq);
119 print $hq . "\n";
120 print $lq . "\n";
121
122
123 if($opt{'h'}){
124     usage();
125     exit 0;
126 }
127
128
129 if($opt{'a'}, $opt{'H'}, $opt{'L'}){
130
131     # aaa
132     print "Option -a given, will now sort all machines in ALTO \n";
133     sleep(3);
```

```
134     getstatus();
135     # getservicestatus();
136     getcomputestatus();
137
138     foreach my $computenode (keys %status) {
139
140         print "$computenode: \n";
141
142         foreach my $vm (keys %{$status{$computenode}}) {
143
144             print "\t vm-name:$vm\n";
145
146             $vcpu_nr = ($status{$computenode}{$vm}{"vcpu"}-1);
147
148             if ($vm ~~ @not_important) {
149                 my $i = 0;
150                 while ($i <= $vcpu_nr){
151                     # system("ssh $node virsh vcpupin $line $i $low_cpu");
152                     # print "\tssh $targetnode virsh vcpupin $vm $i $lq \n";
153                     $commands{$computenode} .= "virsh vcpupin $vm $i $lq; ";
154                     $i++;
155                 }
156
157             }else{
158                 my $i = 0;
159                 while ($i <= $vcpu_nr){
160                     # system("ssh $node virsh vcpupin $line $i $low_cpu");
161                     # print "\tssh $targetnode virsh vcpupin $vm $i $hq \n";
162                     $commands{$computenode} .= "virsh vcpupin $vm $i $hq; ";
163                     $i++;
164                     #  print "command er naa= " . $commands{$targetnode};
165                 }
166             }
167         }
168     }
169
170
171 #    printcommands();
172
173 foreach my $computenode (keys %commands) {
174 #        print "\n\nssh $computenode \"" . $commands{$computenode} . "\"";
175
176     $send_command = "ssh " .  $computenode . " \"" . $commands{$computenode} .
177 "\"";
178
179     print "command:$send_command\n";
180     # system($send_command)
181
182 }
183
184
185
186 ############################ opt -u and -r ##############################
187
188 }elsif($opt{'u'} and $opt{'r'}){
189
190     print "Option -u given, will now VCPU pin $uuid \n";
191     sleep(3);
192     getstatus();
193     # getservicestatus();
194     getcomputestatus();
195 #    printstatus();
196
197     my $uuid = $opt{'u'};
198 # range
199     my $only_range = $opt{'r'};
200     if($opt{'r'}){
201         $only_range =~  (/(\d+)-(\d+)/);
```

```
202         my $only_range_low = $1;
203         my $only_range_high = $2;
204
205 #         $only_range_low = ($only_range_low-1);
206 #         $only_range_high = ($only_range_high-1);
207         # print "range = " . $only_range_low . "\n";
208         # print "range = " . $only_range_high . "\n";
209         for($only_range_low; $only_range_low <= $only_range_high; $only_range_low++){
210             $only_q .= "$only_range_low,";
211         }
212     }
213     # print "Range = " . $only_q;
214     chop($only_q);
215
216     foreach my $computenode (keys %status) {
217 #         print "$computenode: \n";
218         foreach my $vm (keys %{$status{$computenode}}) {
219
220             if($status{$computenode}{$uuid}){
221                 $only_computenode = $computenode;
222                 $only_vm = $vm;
223                 sleep(2);
224                 $vcpu_nr = ($status{$computenode}{$vm}{"vcpu"}-1);
225             }
226         }
227     }
228     print "$only_vm located at $only_computenode, will now set cpu range for
229   number of VCPU = " . ($vcpu_nr+1) . "\n";
230     sleep(2);
231     $only_command = "ssh $only_computenode \"";
232
233     for(my $i= 0; $i <= $vcpu_nr; $i++){
234         $only_command .= "virsh vcpupin $only_vm $i $only_q;";
235         }
236
237     $only_command .= "\"";
238
239     print "Only command = " . $only_command . "\n";
240
241     system($only_command);
242     system("ssh $only_computenode virsh vcpuinfo $only_vm");
243
244
245 }elsif($check==1){
246
247     print "Option -c given, will now check environment\n";
248     sleep(3);
249     getstatus();
250     # getservicestatus();
251     getcomputestatus();
252     # printstatus();
253
254     foreach my $computenode (keys %status) {
255
256         foreach my $vm (keys %{$status{$computenode}}) {
257
258             #    print "\t vm-name:$vm\n";
259
260             $vcpu_nr = ($status{$computenode}{$vm}{"vcpu"}-1);
261
262             $commands{$computenode} .= "virsh vcpuinfo $vm; ";
263
264         }
265     }
266
267     foreach my $computenode (keys %commands) {
268         print "$computenode: \n";
269 #         print "\n\nssh $computenode \"" . $commands{$computenode} . "\"";
```

```
        $send_command = "ssh " .  $computenode . " \"" . $commands{$computenode} .
"\"" . "\| grep Affinity";
#       print "command:$send_command\n";
        system($send_command)
    }
}else{
################################################################################
        print "not valid arguments. -a for all vms, or -u uuid and -r range!\n";
    die;

}



################################################
# Subroutines



sub getstatus {

    # Get the rows from database
    my $dbh = DBI->connect("DBI:mysql:$database;host=$server", $username,
$password)
     || die "Could not connect to database: $DBI::errstr";
    my $sth = $dbh->prepare('select * from instances where vm_state="active";')
     || die "$DBI::errstr";
    $sth->execute();

    # Print number of rows found
    if ($sth->rows < 0) {
        print "Sorry, no vm found failure in getstatus subrutine.\n";
    } else {
        # printf ">> Found something\n", $sth->rows;
        # Loop if results found
        while (my $results = $sth->fetchrow_hashref) {

            my $vm_name = $results->{uuid}; # get the hostname
            my $computenode = $results->{host}; # get the computenode
            my $vm_vcpu = $results->{vcpus}; # VM number of cpu
            my $vm_memory = $results->{memory_mb}; # VM number of memory
            my $vm_project_id = $results->{project_id}; # VM number of memory
        #    my $vm_uuid = $results->{uuid}; # VM uuid


             #   printf  $vm_name . " at ". $computenode . " with VCPU: " . $vm_vcpu .
" and memory: " . $vm_memory . "\n";

            $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
         #   $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
         #   $status{$computenode}{$vm_name}{"project_id"}=$vm_project_id;
         #   $status{$computenode}{$vm_name}{"uuid"}=$vm_uuid;

            $vms{$results->{uuid}} = $computenode;
            $nametable{$results->{uuid}} = $results->{"hostname"};



            if ($vm_name ~~ @not_important) {
                $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
#                 print "NTANT\n"; #  " . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
                $important_vcpu{$computenode}{"not_important_vcpu"} =
($important_vcpu{$computenode}{"not_important_vcpu"} + $vcpu_nr);


            }else{

                $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
```

```
338
339 #                print "IMPORTANT\n"; # . $vm . " Vcpu: " . ($vcpu_nr+1) . "\n";
340                 $important_vcpu{$computenode}{"important_vcpu"} =
341 ($important_vcpu{$computenode}{"important_vcpu"} + $vcpu_nr);
342
343
344             }
345
346
347         }
348
349 # Disconnect
350         $sth->finish;
351         $dbh->disconnect;
352     }
353 }
354
355 sub getcomputestatus {
356
357     # Get the rows from database
358     my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server", $username,
359 $password)
360       || die "Could not connect to database: $DBI::errstr";
361     my $sth2 = $dbh2->prepare('select * from compute_nodes')
362       || die "$DBI::errstr";
363     $sth2->execute();
364
365 # Print number of rows found
366     if ($sth2->rows < 0) {
367         print "Sorry, no vm found failure in getstatus subrutine.\n";
368     } else {
369         # printf ">> Found something\n", $sth->rows;
370         # Loop if results found
371         while (my $results2 = $sth2->fetchrow_hashref) {
372
373
374             my $compute_name = $results2->{hypervisor_hostname}; # get the hostname
375             my $compute_vcpu = $results2->{vcpus}; # VM number of cpu
376             my $compute_memory = $results2->{memory_mb}; # VM number of memory
377             my $compute_vcpu_used = $results2->{vcpus_used}; # VM number of vcpu used
378             my $compute_memory_used = $results2->{memory_mb_used}; # VM number of vcpu
379 used
380             my $compute_running_vms = $results2->{running_vms}; # VM number of vcpu
381 used
382             my $compute_free_memory = $compute_memory - $compute_memory_used;
383             my $compute_free_vcpu = 64 - $compute_vcpu_used;
384
385
386             # print  $compute_name . " running VM: " . $compute_running_vms . " number
387 of used vcpu : " . $compute_vcpu_used . "free VCPU: " . $compute_free_vcpu  .
388 "\n";
389             if ( $computestatus{$compute_name} ){
390                 $computestatus{$compute_name}{"running vms"}=$compute_running_vms;
391                 $computestatus{$compute_name}{"compute_free_vcpu"}=$compute_free_vcpu;
392                 $computestatus{$compute_name}{"compute_free_memory"}=$compute_free_memory;
393
394
395                 $vm_count{$compute_name}=$compute_running_vms;
396                 $compute_receive{$compute_name}=0;
397                 $commands{$compute_name};
398             }
399
400         }
401     }
402
403 # Disconnect
404     $sth2->finish;
405     $dbh2->disconnect;
```

```
406 }
407
408
409 sub printcompute {
410
411     foreach my $computenode (sort(keys %computestatus)) {
412         print "$computenode: \n";
413         foreach my $item (keys %{$computestatus{$computenode}}) {
414             print "\t$item: ".$computestatus{$computenode}{$item}."\n";
415
416         }
417     }
418 }
419
420 sub printimportant {
421
422     foreach my $computenode (sort(keys %important_vcpu)) {
423         print "$computenode: \n";
424         foreach my $item (keys %{$important_vcpu{$computenode}}) {
425             print "\t$item: ".$important_vcpu{$computenode}{$item}."\n";
426
427         }
428     }
429 }
430
431
432
433 sub printstatus {
434     foreach my $computenode (keys %status) {
435         print "$computenode: \n";
436         foreach my $vm (keys %{$status{$computenode}}) {
437             print "\t vm-name:$vm\n";
438             foreach my $noe (keys %{$status{$computenode}{$vm}}) {
439                 print "\t$noe: " . $status{$computenode}{$vm}->{$noe} . "\n";
440
441
442             }
443         }
444     }
445 }
446
447
448
449
450
451 sub getLoggingTime {
452
453     my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
454     my $nice_timestamp = sprintf ( "%04d.%02d.%02d:%02d:%02d:%02d",
455                                    $year+1900,$mon+1,$mday,$hour,$min,$sec);
456     return $nice_timestamp;
457 }
458
459
460 sub usage {
461
462     print "Usage:\n";
463     print "-c to check print cpu affinity at all computenodes\n";
464     print "-a to divide all virtual machines in HQ and LQ, must give -L
465 low_quality as 4-25 and -H high_quality 26-63\n";
466     print "-u for uuid must give -r range for specific uuid\n";
467     print "-h for help\n";
468     print "-v for verbose(more output)\n";
469     print "-d for debug(even more output)\n";
470
471 }
472
473 sub verbose {
```

```
474         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
475     }
476
477     sub debug {
478         print "DEBUG: " . $_[0] if ($DEBUG);
479     }
480
481
482
483
484     sub printcommands {
485        foreach my $computenode (keys %commands) {
486         print "\n\nssh $computenode \"" . $commands{$computenode} . "\"";
487         }
488     }
```

9.5 Monitoring script for Count migrations

countmigrations.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables

my $VERBOSE = 0;
my $DEBUG = 0;
my $MIGRATIONPAUSE = 5;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname
my %status;
my @old_locations;
my @new_locations;
my @migrations;
my $count = 0;
my $timestamp = time;
# command line options

my $opt_string = "vdh";
getopts("$opt_string", \my %opt) or usage() and exit(1);

$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

if($opt{'h'}){
        usage();
        exit 0;
}


##################################################
# Main part

getstatus();


if (-e 'migration_count.txt'){
    open(MYFILE, "<" .  "migration_count.txt");
    foreach my $computenode (keys %status) {
        foreach my $vm (keys %{$status{$computenode}}) {
            push(@new_locations, "$computenode:$vm");
        }
   }
 #   print @new_locations;
#

    foreach my $line (<MYFILE>){
        chomp($line);
        push(@old_locations, $line)
    }

#    print "\n";
 #   print @old_locations;


my @diff = grep{ not $_ ~~ @old_locations } @new_locations;

    if (@diff){
```

```
66          # print "found diff @diff";
67          foreach my $vm (@diff){
68
69              $vm =~ /(compute\d+):(.+-.+-.+-.+)/;
70              my $node = $1;
71              my $uuid = $2;
72            #  print $uuid;
73
74
75              if (grep{/$uuid/} @old_locations){
76                  # print "\n $uuid is migrated new_locations\n";
77                  $count++;
78              }
79
80          }
81
82
83      }
84
85
86  }else{
87      # This is first execute and creating log file for later
88  comparison.
89      open(MYFILE, '>migration_count.txt');
90      foreach my $computenode (keys %status) {
91          foreach my $vm (keys %{$status{$computenode}}) {
92              print MYFILE "$computenode:$vm\n";
93          }
94     }
95  }
96
97  print "openstack.migrations " . $count . " " . $timestamp;
98
99
100 ####################  Updating file #################
101 open(MYFILE, '>migration_count.txt');
102  foreach my $computenode (keys %status) {
103     foreach my $vm (keys %{$status{$computenode}}) {
104         print MYFILE "$computenode:$vm\n";
105     }
106 }
107
108
109
110 close(MYFILE);
111 ###################################################
112 # Subroutines
113
114
115
116 sub getstatus {
117
118     # Get the rows from database
119     my $dbh = DBI->connect("DBI:mysql:$database;host=$server",
120 $username, $password)
121       || die "Could not connect to database: $DBI::errstr";
122     my $sth = $dbh->prepare('select * from instances where
123 vm_state="active";')
124       || die "$DBI::errstr";
125     $sth->execute();
126
127     # Print number of rows found
128     if ($sth->rows < 0) {
129         print "Sorry, no vm found failure in getstatus
130 subrutine.\n";
131     } else {
132         # printf ">> Found something\n", $sth->rows;
133         # Loop if results found
```

```
134         while (my $results = $sth->fetchrow_hashref) {
135
136           my $vm_name = $results->{uuid}; # get the hostname
137           my $computenode = $results->{host}; # get the computenode
138           my $vm_vcpu = $results->{vcpus}; # VM number of cpu
139           my $vm_memory = $results->{memory_mb}; # VM number of
140   memory
141           my $vm_project_id = $results->{project_id}; # VM number
142   of memory
143         #    my $vm_uuid = $results->{uuid}; # VM uuid
144
145
146           #   printf  $vm_name . " at ". $computenode . " with
147   VCPU: " . $vm_vcpu . " and memory: " . $vm_memory . "\n";
148
149           $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
150           $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
151           $status{$computenode}{$vm_name}{"project_id"}=$
152   vm_project_id;
153         #   $status{$computenode}{$vm_name}{"uuid"}=$vm_uuid;
154
155
156         }
157
158   # Disconnect
159         $sth->finish;
160         $dbh->disconnect;
161       }
162   }
163
164
165
166   sub printstatus {
167       foreach my $computenode (keys %status) {
168         # print "$computenode: \n";
169         foreach my $vm (keys %{$status{$computenode}}) {
170           print "$computenode: $vm\n";
171           # foreach my $noe (keys %{$status{$computenode}{$vm}}) {
172         #       $noe: " . $status{$computenode}{$vm}->{$noe} . "\n";
173         #
174
175         #   }
176         }
177       }
178   }
179
180
181
182   sub usage {
183
184         print "Usage:\n";
185         print "-h for help\n";
186         print "-v for verbose(more output)\n";
187         print "-d for debug(even more output)\n";
188
189   }
190
191   sub verbose {
192         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
193   }
194
195   sub debug {
196         print "DEBUG: " . $_[0] if ($DEBUG);
197   }
198
199
200
```

9.6 Monitoring script power consumption

Powerconsumption.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;


# global variables

my $VERBOSE = 0;
my $DEBUG = 0;

# command line options

my $opt_string = "vdh";
getopts("$opt_string", \my %opt) or usage() and exit(1);

$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

if($opt{'h'}){
	usage();
	exit 0;
}


##################################################
# Main part

for (my $i = 1; $i < 12; $i++) {

   # Connect to computenode1 to 11 and ask for powerconsumption info
   my @command = `ssh root\@.$i racadm getconfig -g cfgServerPower`;
   chomp(@command);

   my $timestamp = getLoggingTime();

   open (MYFILE, '>>data.txt');
   foreach my $line (@command){

      $line =~ s/\s//g; # remove whitespace
      $line =~ s/#//g; # remove # in sentence
      $line =~ s/cfgServer//g; # remove cfgServer
      $line =~ s/ACW//g; # remove ACW
      $line =~ s/\|\d{1,7}(B|b)tu\/hr//g; # remove | 3-4 digit followed by Btu/hr

      if($line =~ /Write-Only/){
      }else{
         print MYFILE "energy.server$line $timestamp host=compute$i\n";

         ########### Insert database insert code for each line ###############

         # insert "energy.server$line $timestamp host=compute$i into database
openstack or something


         ####################################################################

      }
   }

   print "Info received from node: " . $i . "\n";
}
close (MYFILE);
```

```
66
67  ##################################################
68  # Subroutines
69
70  sub getLoggingTime {
71
72      my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
73      my $nice_timestamp = sprintf ( "%04d.%02d.%02d:%02d:%02d:%02d",
74                                     $year+1900,$mon+1,$mday,$hour,$min,$sec);
75      return $nice_timestamp;
76  }
77
78  sub usage {
79
80          print "Usage:\n";
81          print "-h for help\n";
82          print "-v for verbose(more output)\n";
83          print "-d for debug(even more output)\n";
84
85  }
86
87  sub verbose {
88          print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
89  }
90
91  sub debug {
92          print "DEBUG: " . $_[0] if ($DEBUG);
93  }
94
```

9.7 Monitoring script for virtual machines

virtualmachinesmonitoring.pl

```
#!/usr/bin/perl
my $tag = " cloud=$ARGV[0]" if $ARGV[0];
my $dbhost = "";
my $prefix = "openstack.vms.";
my $timestamp = time;
# build hash of instance names
use DBI;
my $idtags;
my %serverslookup;
my %instances;
my %projects;
# my $dbh =
DBI->connect("DBI:mysql:nova;host=;port=3360","","") or
warn("error: $!\n");
# if ( $dbh ){

#     my $query = $dbh->prepare("select uuid,hostname from instances");
#     $query->execute();
#     while ( my @results = $query->fetchrow_array() ){
#         $serverslookup{$results[0]} = $results[1];
#         $serverslookup{$results[0]} =~ s/ /_/g;
#     }
# }

my @dbresults = `echo "select uuid,hostname,host,project_id,root_gb from instances
where vm_state != 'deleted'" | mysql -u  --password= -h
IP nova`;
my @keystoneresults = `echo "select id,name" | mysql -u
 keystone`;

foreach my $result (@dbresults){
#    print $result;
    my @ra = split /\s+/,$result;
    $serverslookup{$ra[0]}{'name'} = $ra[1];
    $serverslookup{$ra[0]}{'host'} = $ra[2];
    $serverslookup{$ra[0]}{'project_id'} = $ra[3];
    $serverslookup{$ra[0]}{'root_gb'} = $ra[4];
    $serverslookup{$ra[0]}{'name'} =~ s/ /_/g;
}

foreach my $result (@keystoneresults){
#    print $result;
    my @ra = split /\s+/,$result;
    $projects{$ra[0]} = $ra[1];
}


open(LIST,"virsh list |");
<LIST>;
<LIST>;
while( my $instance = <LIST> ){
    chomp $instance;
    $instance =~ s/^\s+(\d+)\s+.*/$1/;
    next unless $instance;
#    print "instance: '$instance'\n";


# get uid
    my $uuid;
#    my $idtags;
    open(INFO,"virsh dominfo $instance |");
    while ( my $infoline = <INFO> ){
        chomp $infoline;
#        print "infoline: $infoline\n";
        if ( $infoline =~ /UUID:\s+(\S+)$/ ){
```

```
66  #           print "UUID is $1\n";
67              $uuid = $1;
68              $idtags = "name=$serverslookup{$uuid}{'name'}
69  project=$projects{$serverslookup{$uuid}{'project_id'}}
70  host=$serverslookup{$uuid}{'host'} uuid=$uuid";
71  #           $instances{$uuid}{"name"} = $serverslookup{$uuid};
72          } elsif ( $infoline =~ /CPU time:\s+(\d+\.\d)s/ ){
73              out("cpu_time_seconds",$1);
74          }
75      }
76
77      out("disk.root_gb",$serverslookup{$uuid}{'root_gb'});
78
79      # get blklist
80
81      open(BLKLIST,"virsh domblklist $instance |");
82      <BLKLIST>;
83      <BLKLIST>;
84      while( my $blkline = <BLKLIST> ){
85          $blkline =~ /^(\S+)\s/;
86          $disk = $1;
87          if( $disk =~ /vd/ ){
88  #           print "disk: $disk\n";
89              open(BLKSTAT,"virsh domblkstat $instance $disk |");
90              while( my $blkstat = <BLKSTAT> ){
91                  chomp $blkstat;
92                  if ( $blkstat ){
93                      my @ba = split /\s+/,$blkstat;
94                      out("disk.$ba[1]",$ba[2],"disk=$disk");
95                  }
96              }
97              close(BLKSTAT);
98          }
99      }
100     close(BLKLIST);
101
102     # get iflist
103     open(IFLIST,"virsh domiflist $instance |");
104     <IFLIST>;
105     <IFLIST>;
106     while( my $ifline = <IFLIST> ){
107         $ifline =~ /^(\S+)\s/;
108         my $if = $1;
109 #       print $if;
110 #       print "disk: $disk\n";
111         if ( $if =~ /tap/ ){
112             open(IFSTAT,"virsh domifstat $instance $if |");
113             while( my $ifstat = <IFSTAT> ){
114                 chomp $ifstat;
115                 if ( $ifstat ){
116                     my @ia = split /\s+/,$ifstat;
117                 out("network.$ia[1]",$ia[2],"iface=$if");
118                 }
119             }
120             close(IFSTAT);
121         }
122     }
123     close(IFLIST);
124     open(MEMSTAT,"virsh dommemstat $instance |");
125     while ( my $memline = <MEMSTAT> ){
126         chomp $memline;
127         if ( $memline ){
128             ( my $k,my $v ) = split /\s+/,$memline;
129             out("memory.$k",$v); # broken
130         }
131     }
132
133     open(CPU,"virsh cpu-stats $instance |");
```

```
134     while (my $cpuline = <CPU> ){
135
136         if ( $cpuline =~ /(CPU\d+):/ ){
137             my $cpu = $1;
138             my $cputime = <CPU>;
139             my @ca = split /\s+/,$cputime;
140     #           out("cpu.cpu_time_seconds",$ca[2],"cpu=$cpu");
141             my $cputime = <CPU>;
142             my @ca = split /\s+/,$cputime;
143     #           out("cpu.vcpu_time_seconds",$ca[2],"cpu=$cpu");
144         } elsif ( $cpuline =~ /Total:/ ){
145             my $cputime = <CPU>;
146             my @ca = split /\s+/,$cputime;
147             out("cpu.total_cpu_time_seconds",$ca[2]);
148             my $cputime = <CPU>;
149             my @ca = split /\s+/,$cputime;
150             out("cpu.total_user_time_seconds",$ca[2]);
151             my $cputime = <CPU>;
152             my @ca = split /\s+/,$cputime;
153             out("cpu.total_system_time_seconds",$ca[2]);
154
155
156         }
157
158     }
159
160 }
161 close(LIST);
162
163 sub out {
164     print $prefix . "$_[0] $_[1] $timestamp $tag $idtags $_[2]\n";
165 }
166
```

9.8 Monitoring script for compute nodes

computemonitoring.pl

```
#!/usr/bin/perl

# our needed packages

use strict "vars";
use Getopt::Std;
use DBI;

# global variables

my $VERBOSE = 0;
my $DEBUG = 0;
my $MIGRATIONPAUSE = 5;
my $username = ''; # MySQL username
my $password = ''; # MySQL password
my $database = ''; #  MySQL database name
my $server = ''; # server hostname
my %status;
my %computestatus;
my %vm_count;
my %compute_receive;
my %vms;
my %vms_old;
my %nametable;
my %commands;
my %important_vcpu;
my $maxrunningcompute = 0;
my $buffer = 1;
my @running_compute;
my @stopped_compute;
my @error_compute;
my @not_important;
my $targetnode;
my $vacanter;
my $vcpu_subtract;
my $vcpu_nr;
my $livemigration;
# my $important_vcpu;
# my $not_important_vcpu;

open(MYFILE, 'notimportant.txt');
my $timestamp = time();

##### Importance levels ########

foreach my $line (<MYFILE>){

    chomp($line);

     push(@not_important, $line)

}

close(MYFILE);


my $high_cpu =
"5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27
,28,29,30,31,32,33,34,35,36,37,38,39,40";
my $low_cpu =
"41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,
62,63,64";

# command line options

```

```
66  my $opt_string = "vdh";
67  getopts("$opt_string", \my %opt) or usage() and exit(1);
68
69  $VERBOSE = 1 if $opt{'v'};
70  $DEBUG = 1 if $opt{'d'};
71
72  if($opt{'h'}){
73          usage();
74          exit 0;
75  }
76
77
78  ##################################################
79  # Main part
80
81  # getstatus();
82  getcomputestatus();
83  getservicestatus();
84
85  printcompute();
86
87  ######################################## CPU allocation
88  ########################
89
90
91
92  ##################################################
93  # Subroutines
94  sub scaleservers {
95  my $crosspoint = $_[0];
96
97      print "Crosspoint was at $crosspoint\n";
98      # if crosspoint + buffer < maxrunning
99      if( ($crosspoint + $buffer) < $maxrunningcompute){
100         print "We are using too many compute nodes, scaling down\n";
101         for ( my $i = ( $crosspoint + $buffer + 1 ); $i <=
102 $maxrunningcompute; $i++ ){
103             my $number = $i;
104             $number = "0" . $number if $number < 10;
105             my $computename = "compute$number";
106             print "shutting down $computename\n";
107             system("ssh $computename service nova-compute stop");
108         }
109     # open(SSH,"ssh controller nova-manage service list | ");
110
111     }elsif(($crosspoint + $buffer) > $maxrunningcompute){
112         print "we are under capacity, scaling up\n";
113         my $number = $maxrunningcompute + 1;
114         $number = "0" . $number if $number < 10;
115         my $computename = "compute$number";
116         print "starting $computename\n";
117         system("ssh $computename service nova-compute start");
118
119     }
120 }
121
122
123 sub getVMLocation {
124     my $vm = $_[0];
125     my $dbh = DBI->connect("DBI:mysql:$database;host=$server",
126 $username, $password)
127       || die "Could not connect to database: $DBI::errstr";
128     my $sth = $dbh->prepare("select * from instances where uuid
129 ='$vm';")
130       || die "$DBI::errstr";
131     $sth->execute();
132     my $results = $sth->fetchrow_hashref;
133     $sth->finish;
```

```
134     $dbh->disconnect;
135
136     return $results->{"host"};
137
138 }
139
140 sub getstatus {
141
142     # Get the rows from database
143     my $dbh = DBI->connect("DBI:mysql:$database;host=$server",
144 $username, $password)
145       || die "Could not connect to database: $DBI::errstr";
146     my $sth = $dbh->prepare('select * from instances where
147 vm_state="active";')
148       || die "$DBI::errstr";
149     $sth->execute();
150
151     # Print number of rows found
152     if ($sth->rows < 0) {
153         print "Sorry, no vm found failure in getstatus
154 subrutine.\n";
155     } else {
156         # printf ">> Found something\n", $sth->rows;
157         # Loop if results found
158         while (my $results = $sth->fetchrow_hashref) {
159
160             my $vm_name = $results->{uuid}; # get the hostname
161             my $computenode = $results->{host}; # get the computenode
162             my $vm_vcpu = $results->{vcpus}; # VM number of cpu
163             my $vm_memory = $results->{memory_mb}; # VM number of
164 memory
165             my $vm_project_id = $results->{project_id}; # VM number
166 of memory
167         #    my $vm_uuid = $results->{uuid}; # VM uuid
168
169
170             #   printf  $vm_name . " at ". $computenode . " with
171 VCPU: " . $vm_vcpu . " and memory: " . $vm_memory . "\n";
172
173             $status{$computenode}{$vm_name}{"vcpu"}=$vm_vcpu;
174             $status{$computenode}{$vm_name}{"memory"}=$vm_memory;
175             $status{$computenode}{$vm_name}{"project_id"}=$
176 vm_project_id;
177         #   $status{$computenode}{$vm_name}{"uuid"}=$vm_uuid;
178
179             $vms{$results->{uuid}} = $computenode;
180             $nametable{$results->{uuid}} = $results->{"hostname"};
181
182
183
184             if ($vm_name ~~ @not_important) {
185                 $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
186
187                 $important_vcpu{$computenode}{"not_important_vcpu"} =
188 ($important_vcpu{$computenode}{"not_important_vcpu"} +
189 $vcpu_nr);
190
191
192             }else{
193
194                 $vcpu_nr = ($status{$computenode}{$vm_name}{"vcpu"});
195
196                 #  print "IMPORTANT:  " . $vm . " Vcpu: " . ($vcpu_nr+1)
197 . "\n";
198                  $important_vcpu{$computenode}{"important_vcpu"} =
199 ($important_vcpu{$computenode}{"important_vcpu"} + $vcpu_nr);
200
201
```

```
202             }
203
204
205         }
206
207     # Disconnect
208         $sth->finish;
209         $dbh->disconnect;
210     }
211     }
212
213     sub getcomputestatus {
214
215         # Get the rows from database
216         my $dbh2 = DBI->connect("DBI:mysql:$database;host=$server",
217     $username, $password)
218           || die "Could not connect to database: $DBI::errstr";
219         my $sth2 = $dbh2->prepare('select * from compute_nodes')
220           || die "$DBI::errstr";
221         $sth2->execute();
222
223     # Print number of rows found
224         if ($sth2->rows < 0) {
225             print "Sorry, no vm found failure in getstatus
226     subrutine.\n";
227         } else {
228             # printf ">> Found something\n", $sth->rows;
229             # Loop if results found
230             while (my $results2 = $sth2->fetchrow_hashref) {
231
232
233                 my $compute_name = $results2->{hypervisor_hostname}; #
234     get the hostname
235                 my $compute_vcpu = $results2->{vcpus}; # VM number of
236     cpu
237                 my $compute_memory = $results2->{memory_mb}; # VM number
238     of memory
239                 my $compute_vcpu_used = $results2->{vcpus_used}; # VM
240     number of vcpu used
241                 my $compute_memory_used = $results2->{memory_mb_used}; #
242     VM number of vcpu used
243                 my $compute_running_vms = $results2->{running_vms}; # VM
244     number of vcpu used
245                 my $compute_free_memory = $compute_memory -
246     $compute_memory_used;
247                 my $compute_free_vcpu = 64 - $compute_vcpu_used;
248
249
250                 # print  $compute_name . " running VM: " .
251     $compute_running_vms . " number of used vcpu : " .
252     $compute_vcpu_used . "free VCPU: " . $compute_free_vcpu  . "\n";
253
254                 $computestatus{$compute_name}{"running
255     vms"}=$compute_running_vms;
256                 $computestatus{$compute_name}{"compute_free_vcpu"}=$
257     compute_free_vcpu;
258                 $computestatus{$compute_name}{"compute_free_memory"}=$
259     compute_free_memory;
260
261
262                 $vm_count{$compute_name}=$compute_running_vms;
263                 $compute_receive{$compute_name}=0;
264                 $commands{$compute_name};
265
266
267             }
268         }
269
```

```
# Disconnect
    $sth2->finish;
    $dbh2->disconnect;
}


sub printcompute {

    my $prefix = "openstack.compute.";
    foreach my $computenode (sort(keys %computestatus)) {
#       print "$computenode: \n";

        # face :-) host=compute11
      # compute_free_vcpu 41 host=compute11
      # status enabled host=compute11
      # compute_free_memory 220050 host=compute11
      # running vms 11 host=compute11
        # foreach my $item (keys %{$computestatus{$computenode}}) {
        #     print "$item ".$computestatus{$computenode}{$item}."
host=$computenode\n";
        #  }
        print "${prefix}compute_free_mem
".$computestatus{$computenode}{"compute_free_memory"}."
$timestamp host=$computenode\n";
        print "${prefix}compute_free_vcpu
".$computestatus{$computenode}{"compute_free_vcpu"}." $timestamp
host=$computenode\n";
        my $enabled = 0;
        $enabled = 1 if ( $computestatus{$computenode}{"face"} eq
":-)" and $computestatus{$computenode}{"enabled"} eq "enabled"
);
        print "${prefix}enabled $enabled $timestamp
host=$computenode\n";


        # }
    }
}

sub printimportant {

    foreach my $computenode (sort(keys %important_vcpu)) {
        print "$computenode: \n";
        foreach my $item (keys %{$important_vcpu{$computenode}}) {
            print "\t$item:
".$important_vcpu{$computenode}{$item}."\n";

        }
    }
}



sub printstatus {
    foreach my $computenode (keys %status) {
        print "$computenode: \n";
        foreach my $vm (keys %{$status{$computenode}}) {
            print "\t vm-name:$vm\n";
            foreach my $noe (keys %{$status{$computenode}{$vm}}) {
                print "\t$noe: " . $status{$computenode}{$vm}->{$noe} .
"\n";


            }
        }
    }
}

```

```
338
339
340
341
342 sub getLoggingTime {
343
344     my
345 ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(
346 time);
347     my $nice_timestamp = sprintf (
348 "%04d.%02d.%02d:%02d:%02d:%02d",
349                                 $year+1900,$mon+1,$mday,$hour
350 ,$min,$sec);
351     return $nice_timestamp;
352 }
353
354
355 sub usage {
356
357         print "Usage:\n";
358         print "-h for help\n";
359         print "-v for verbose(more output)\n";
360         print "-d for debug(even more output)\n";
361
362 }
363
364 sub verbose {
365         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
366 }
367
368 sub debug {
369         print "DEBUG: " . $_[0] if ($DEBUG);
370 }
371
372
373
374 sub getservicestatus {
375
376     # nova-compute     compute09                          nova
377           enabled    :-)   2014-03-21 19:51:32
378     open(SSH,"ssh controller nova-manage service list | ");
379     while( my $line = <SSH> ){
380         if ( $line =~
381 /\s*nova-compute\s+(\S+)\s+nova\s+(\S+)\s+(\S+)\s+\d/ ){
382             $computestatus{$1}{"status"} = $2;
383             $computestatus{$1}{"face"} = $3;
384             my $compute_name = $1;
385             $compute_name =~ /[a-z]+0*(\d+)$/;
386             my $compute_number = $1;
387             $computestatus{$compute_name}{"compute_number"}=$
388 compute_number;
389             $maxrunningcompute = $compute_number if (
390 $compute_number > $maxrunningcompute and
391 $computestatus{$compute_name}{"face"} eq ":-)" and
392 $computestatus{$compute_name}{"status"} eq "enabled");
393         }
394     }
395
396 }
397
398 sub runMigrations {
399
400     foreach my $vm ( keys %vms){
401
402         if ( $vms{$vm} ne $vms_old{$vm} ){
403             print "Migrating $nametable{$vm}: $vms_old{$vm} ->
404 $vms{$vm}\n";
405             # ssh -t controller "source creds; nova live-migration
```

```
406 ad1e2027-5eca-492a-ad3e-872fc086e7dd compute06
407             if ( $nametable{$vm} =~ /fiobench/ ){
408                 print "All systems GO!\n";
409                 system("ssh -t controller 'source creds; nova
410 live-migration $vm $vms{$vm}'");
411                 while( getVMLocation($vm) ne $vms{$vm}){
412                     verbose("sleeping...\n");
413                     sleep $MIGRATIONPAUSE;
414                 }
415             }
416         }
417
418     }
419
420 }
421
422
423 sub printcommands {
424     foreach my $computenode (keys %commands) {
425         print "\n\nssh $computenode " . $commands{$computenode};
426         }
427 }
428
429 sub countmigrations {
430
431     foreach my $computenode (keys %compute_receive) {
432         $livemigration = ($livemigration +
433 $compute_receive{$computenode});
434     }
435
436     print "\n\nThe number of live migration is: " .
437 $livemigration . "\n";
438 }
```